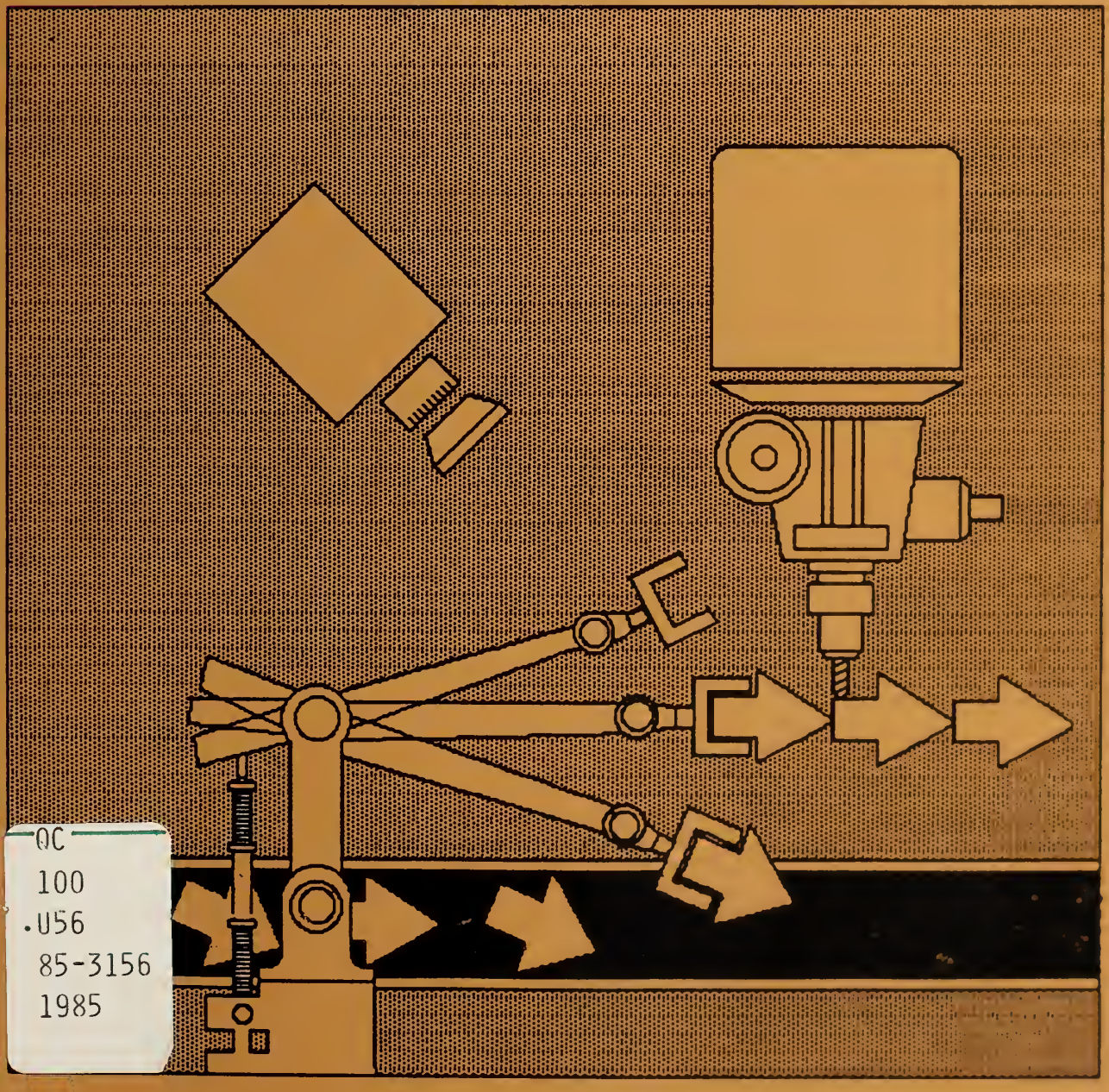
U. S. Department
of Commerce

National Bureau
of Standards

NBS-IR-85-3156

Hierarchical Control System Emulation
User's Manual

January 1985



NBS-IR-85-3156

Hierarchical Control System Emulation User's Manual

Cita M. Furlani, Editor

January 1985

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Factory Automation Systems Division
Gaithersburg, MD 20899

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acknowledgement

The major part of the work that is represented in this manual was done by Bolt Beranek and Newman, Inc. under Department of Commerce Contract NB81SBCA0826 entitled "Emulation/Simulation of an Automatic Manufacturing Test Facility". This manual in its original form was derived from the final report (dated October 1982) from that contract. The product has been further modified at the National Bureau of Standards. This manual documents the state of the Hierarchical Control Systems Emulator as of January 1985.

1.0 INTRODUCTION

1.1 Problem Overview

The design of any feedback control system with even modest complexity requires simulation capability. The controlled plant and feedback law are simulated in order to assess the possible effects of discrepancies between the design model of the plant and the actual plant, and in order to verify the correctness of design approximations. In feedback systems, the effects of such discrepancies can be particularly serious and may be manifested as classical instability in continuous-state systems, or as errors (trap states) and/or loss of synchronization in discrete-state systems. Simulation averts the most serious errors of these types because the control system design can be corrected without physical damage to the actual plant. Most simulations today are digital rather than analog.

Increasingly, the implementation of control systems is also digital rather than analog, and this raises some new issues:

1. What capacity and speed of computer is required?
2. What is the impact of computing and communication delays on closed-loop system performance?
3. What structure of real-time software should be employed?
4. What is the impact of errors in programming logic, or of errors in the designer's conceptual model of the system's logical relationships?
5. What is the impact of discrete failures or of software failures?

In order to answer these questions, it is not sufficient to merely simulate the input-output relationships of the control system. Rather, the internal logical structure and implementation features of the control system (as well as the controlled system itself) need to be represented. Ideally, a one-for-one representation of the control system logic and timing is desired for this purpose. The term emulation* has been applied to this sort of "one-for-one" simulation.

In contrast to methods for designing feedback control of continuous-state dynamic systems, analytic methods for the design of discrete-state feedback systems are at a very primitive stage. Since the available design methods are largely heuristic, even more emphasis must be placed on the iterative design process whereby an initial design is improved on the basis of simulation results. A rather general heuristic for the design of discrete control systems is the modular hierarchical approach proposed by Albus, Barbera and Nagel (1980)**. The Hierarchical Control System Emulator (HCSE) described herein provides emulation/simulation capabilities for a broad class of systems -- in particular, automated manufacturing systems -- which can be controlled by modular hierarchical control systems.

---

* The term came into use in the context of verification of computer software, where one computer was made to emulate the operation of another which typically was in the design stage. The operating system software, which is designed this way, actually implements a discrete feedback control system.

** See Section 2.1 for further details of this approach.

1.2 Solution Overview

The use of the Hierarchical Control System Emulation is described in Sections 3-5 of this manual, and the more eager user is referred directly to those sections. The operation of the emulation may be divided into three phases:

1. Data entry (making modules)
2. Running the HCSE
3. Data logging and analysis

The user enters data for the emulation/simulation entirely by writing module descriptions. Each module has a common format based on a generalized state-machine description with named variables. Communication between modules, which is transparent to the user, is achieved totally through storage (by name) of common input and output variables in a shared (common) memory; access to the memory is time-slice synchronized.* The pattern and sequencing of input/output transactions between modules may be specified by the user to define a hierarchical relationship of the control system modules. The module format is sufficiently general that modules may be used to simulate physical devices (subsystems of the controlled plant) as well as emulating control system components. Normally, these simulation modules are at the bottom of the hierarchy and communicate horizontally with each

---

* The module format and shared memory implementation were dictated by requirements of the NBS Automated Manufacturing Research Facility.

other as well as vertically with the hierarchical control system (Figure 1).

In order to run the emulation, the modules may be combined into subsets which are translated into executable form and run as independent processes. These processes are synchronized through common memory. The run-time display runs as another process and allows the user to monitor the real-time progress of the emulation. The user may synchronize the actual rate of progress of the emulation through the run-time display to achieve single-cycle operation, wall-clock sychronization with variable time-scaling, or free-running (maximum-speed) emulation. The user selects the variables from common memory which are to be displayed, and may stop the emulation to record "snapshots" of common memory at any time.

Prior to running the emulation, the user may select certain variables for "logging" purposes. Upon completion of a run, these logging files may be processed to produce summary statistics concerning the values taken by each logged variable and the amount of time spent at each value.

The emulation meets a number of key requirements for hierarchical control system design and evaluation. Communication and computing delays can be emulated. The allocation of modules to different physical processors can be emulated. Different choices and allocations of module functions can be evaluated, and the effects of coding changes within modules can be assessed.

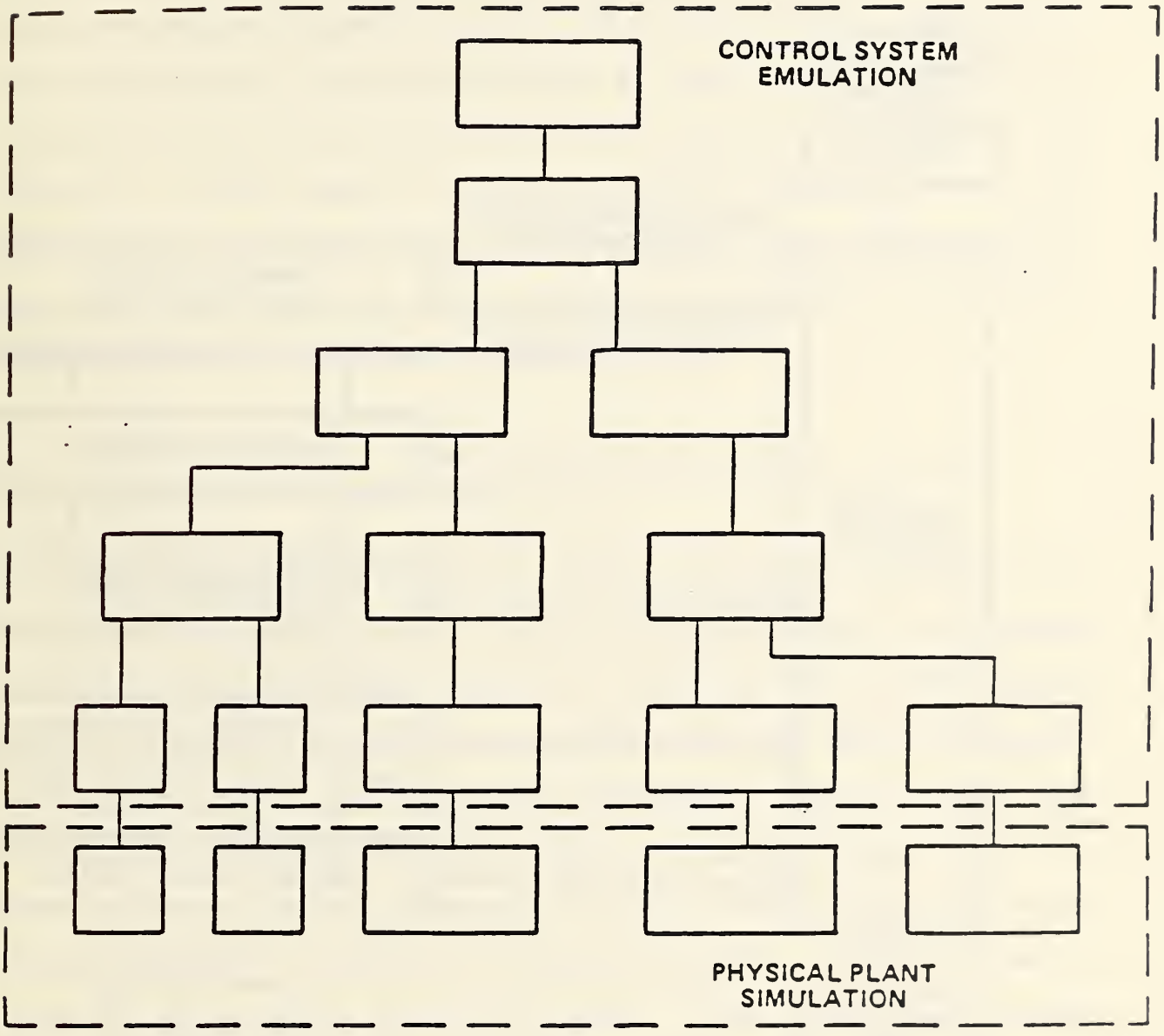


FIG. 1 TYPICAL MODEL STRUCTURE

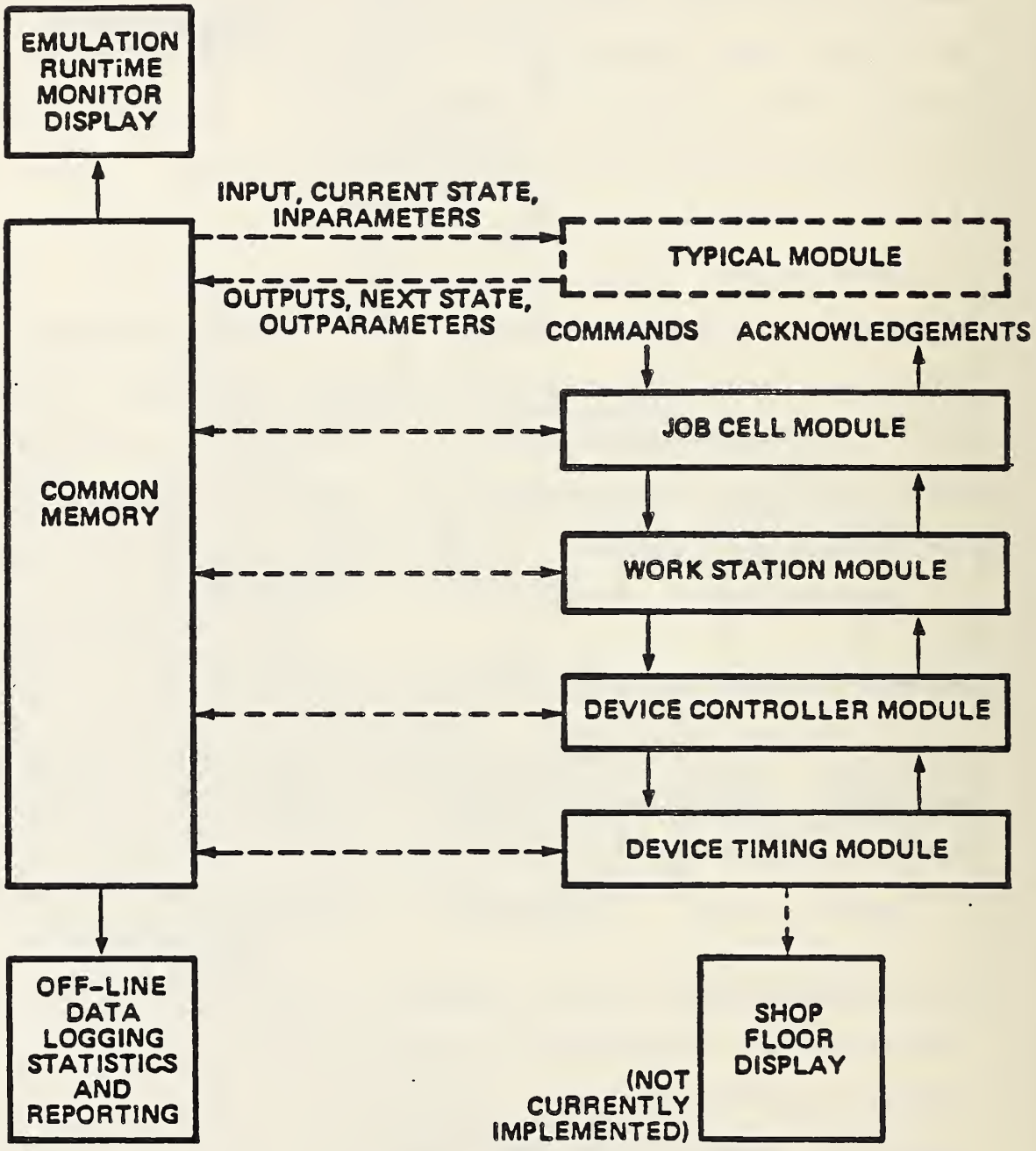


FIG. 2 EMULATION STRUCTURE

Both continuous and discrete variables can be represented; in particular, both continuous physical processes and decision-making processes may be included in the evaluation. Both discrete and continuous sources of error may be represented. Finally, the level of detail in the emulation is under the control of the user, so that critical operations may be represented with great accuracy, while only rough approximations of other subsystems are used. These and additional capabilities are discussed in more detail in Section 7.

1.3 Praxis And Related Documentation

The hierarchical control system emulation is written primarily in Praxis -- a modern, strongly-typed structured language developed by BBN, Inc. -- to run on a VAX 11/780* Digital Equipment Corporation processor with VMS operating system and DEC-supported terminal (or equivalent, such as the BBN Bitgraph). During development, certain portions of the emulation were written in Ratfor (Rational Fortran). This code has been translated into Fortran 77 and into Praxis. Extensive use is made of VMS operating system utilities. The references listed below provide appropriate background in these areas. The reader is assumed to possess elementary knowledge of the VMS operating system and the Praxis language. Section 8.3 contains a Praxis primer.

---

* Certain commercial products are identified in this manual in order to adequately describe the HCSE. Such identification does not imply recommendation or endorsement by the National Bureau of Standards.

The purpose of this manual is to describe the operational aspects of the HCSE completely. Thus, it is a self-contained reference for the user. In addition, the HCSE Applications Guide describes in detail the substantial emulation example modeled after a portion of the NBS Automated Manufacturing Research Facility.* The Programmer's Manual describes the emulation software itself and is intended for use by those who wish to maintain, augment, or modify the HCSE.

Reference List

1. VAX/VMS Command language User's Guide, Digital Equipment Corportation, Maynard, MA., 1980.

2. Praxis Language Reference Manual, BBN Report 4582, January 1981 (see also directory [PRAXIS.DOC] for on-line documentation).

3. Johnson, T.L., Milligan, S.D. and Fortmann, T.E., "Hierarchical Control System Emulation User's Manual", BBN report No. 5096, Bolt Beranek and Newman, Inc., Cambridge, MA., July 1982.

4. Johnson, T.L., Milligan, S.D. and Fortmann, T.E., "Hierarchical Control System Emulation Applications Guide", BBN report No. 5094, Bolt Beranek and Newman, Inc., Cambridge, MA., July 1982.

5. Milligan, S.D.,Johnson, T.L., and Fortmann, T.E., "Hierarchical Control System Emulation Programmer's Manual", BBN Report No. 5095, Bolt Beranek and Newman, Inc., Cambridge, MA., July 1982.

---

* The example does not reflect actual hardware or software of the AMRF, as these were not yet fully specified at the time the HCSE was developed.

2.0 BACKGROUND

The purpose of this section is to review the key concepts of hierarchy, modularity, state machine tables, shared memory, and synchronization which have dictated the central features of the emulation. The user will quickly discover that the actual emulation software imposes very few constraints due to these design requirements, so that almost any sort of control system and controlled plant can be represented with relative ease. Nevertheless, the use of the emulation is most convenient when it is consistent with the underlying design concepts. The specific implementation of the key concepts is described in Section 4.

2.1 Modular Hierarchical Real-Time Control Systems

The general concepts of hierarchical control are described in Albus, Barbera and Nagel (1980). An illustration of a hierarchical control structure is shown in Figure 3. When a command is entered at the top of the hierarchy, it is successively decoded (or interpreted) into more and more detailed instructions at the lower levels of the hierarchy, until the lowest levels of the hierarchy provide an interface with the physical process being controlled. This accounts for information propagation down the hierarchy.* The sensory-interactive hierarchy also provides for the crucial upward flow of sensory

---

* The diagram does not depict other data flows, such as data base interfaces.

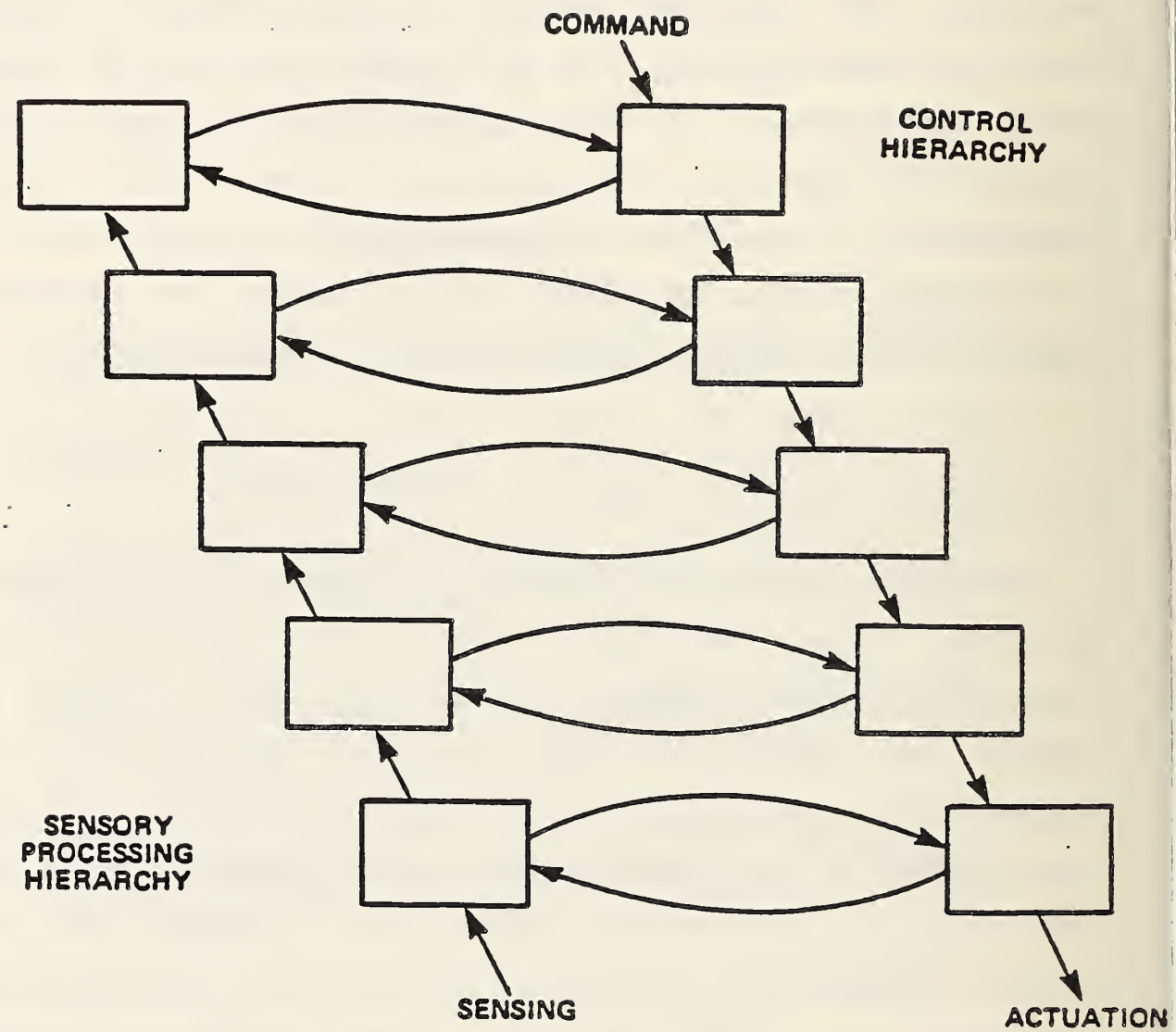


FIG. 3. HIERARCHICAL CONTROL STRUCTURE

information about the physical process being controlled; this information is abstracted (digested) as it passes upward in the hierarchy. Each level of the hierarchy must reconcile its commands from above with the actual state of events below in order to send appropriate sub-commands or corrective actions to the level below it. Such a system is most efficient when errors are handled on the lowest level at which there is command information that is sufficient to formulate appropriate corrective action: "local" errors are corrected at lower levels, while "global" errors are corrected at higher levels. This explicit use of feedback at all levels of the hierarchy and the hierarchical decoding of commands distinguish the sensory-interactive hierarchical approach from the more traditional preplanning approach where errors typically cause emergency shutdowns which necessitate complete replanning or rescheduling. Feedback occurs at all levels of the sensory-interactive hierarchy.

The basic functions of a typical module can be summarized as follows:

(1) Interpret command inputs from the level above.

(2) React to acknowledgements and other sensory information passed up from the level below.

(3) Generate subcommands for the level below.

(4) Generate acknowledgements and pass digested sensory data to the level above.

As viewed from a given module, note that (1) and (2) appear as inputs while (3) and (4) appear as outputs. In addition, each module must provide (at least) for the appropriate time-sequencing of its operations; i.e., it must maintain responsiveness to its inputs without sacrificing continuity or fidelity of its output computations.*

In order to comprehend the operation of such sensory-interactive modules and thus to facilitate design and testing of the whole hierarchical control system, it is desirable to impose a discipline on the internal structure of each module. A generalized finite state machine structure is both simple and sufficiently general for this purpose.

2.2 State Machine Concepts

Each module is assumed to be represented by a generalized finite state machine. The module samples its inputs and produces new outputs at every cycle. This assures that input changes will not be unintentionally ignored; that the data to maintain responsiveness is available; and that the outputs are available on every cycle. The response latency thus depends on the (worst-case) cycle time of the machine. The operation of the machine is broken down as follows:

---

* This is a classical problem in the design of any real-time system, and in particular, computer operating systems.

1. Sample current inputs
2. Compute next state
3. Compute next output

The formal definition of a finite-state-sequential machine* requires that the set of possible input values be finite, that the set of states and output values be finite, and that the next-state and read-out functions be representable as boolean functions. Taken literally, these conditions are too restrictive for the class of applications we wish to consider. The generalized state machine concept that is employed in the HCSE retains the elements of the formal definition but broadens the interpretation in the following ways:

1. Inputs and outputs may include real-valued variables and integers.
2. Next-state and read-out functions are "effectively computable", i.e., they may be implemented by subroutines or procedure calls.

The term "threshold finite automation" has been applied to this generalized state machine. Note that the dimension of the state set is still finite (in the HCSE it is approximately A**31 where A~50, so this restriction is not too significant in practice).

The implementation of state machines is somewhat simplified by adopting the convention that the next state and output remain unchanged unless one of a (usually small) set of conditions in

---

* Arbib, M.A., and Bobrow, L.S., Discrete Mathematics, W.B. Saunders Co., Philadelphia, PA., 1974.

the current state and inputs occurs. When the current state is given, only a modest number of transition conditions (or "trigger conditions") need to be tested in order to determine the next state. This observation greatly reduces actual computing time; however, it is often the case in practice that although the trigger conditions involve only a small number of inputs variables, the actual computation of the next state and output function may also involve a larger set of other variables that never in themselves cause a state transition. In a strict sense, these variables must also be regarded as inputs; in the HCSE, the term "input" is intended for those input variables which appear in the trigger conditions, while "input parameters" are intended to apply to the non transition-causing inputs. Similarly, "outputs" of a module are seen as triggering state transitions in other modules, while "output parameters" provide non transition-causing variables to other modules. This distinction is purely a matter of style, and no internal distinction is made within each pair of terms in the HCSE.

In the general concept of a state machine, we are mostly concerned with deriving proper input and output values for a particular machine. However, a certain amount of internal processing must take place before output values can be derived from the inputs. The variable "internal" is intended to accomodate such internal processing by retaining the intermediate values which represent the internal state of a module. An "internal" variable is neither an input nor an output from the

module in which it resides. Therefore, its value is totally transparent and inaccessible to the other modules in the emulation. Given this, "internal" variables should only be used whenever the value of a variable is not to be transported beyond the confines of the parent module, i.e., an internal counter. Remember that a module only goes through its state table when its inputs change. Thus, if an internal variable is all that has changed on a particular clock tick, the FSM will remain 'asleep'.

Another variable that accomodates internal processing is the "state" variable. However, a "state" variable is both output and input to the FSM that declares it. The "state" variable may be used to monitor the internal process of a module.

State machines can be specified in a variety of ways and in general, there exist a large number of equivalent ways to implement the same module function. Usually, a module with a smaller state set and fewer input and output variables will be more efficient than one with more input, output, and state values (here, we are referring to the states of the machine, not "state" variables.) in the sense that fewer transition conditions need be tested. However, in some cases the next-state and output computations may be simplified when more input, output, or state values are used. For these reasons, it is inadvisable to become particularly attached to a specific module description. Three common methods of specifying state machines are through state transition diagrams, state tables, and ladder diagrams. Conversion of these formats into HCSE form is reviewed in Section

8.2.

2.3 Shared Memory Synchronization

The shared memory contains the current values of all variables which are shared between modules. These include (by convention) the current state* of each module, its input, state, and output variables, and input and output parameters. In order for the common memory to serve as a communication exchange between modules, read-write and overwrite conflicts must be avoided. One way to do this is time-slice synchronization. A fixed time-step is chosen (usually based on the maximum bandwidth requirements of the system). Each time-step or "tick" is subdivided into a read and a write cycle. During the read cycle, each module that requests memory access is permitted to read all of its inputs, input parameters, and state variables and no module may write into common memory. During the write cycle, any module which is ready to write is permitted to write all of its state variables, outputs, and output parameters, but no module may read from common memory.

The emulation achieves this effect in a way that is general enough so that the user retains considerable control over its

---

* This is the "macro-state" of the module; it is not the complete state in a rigorous sense because other local information (e.g., in the computational procedures) is retained between time-steps in order for the module to proceed, in the form of internal variables that are not shared.

real-time performance. All modules which are scheduled to read from memory on a given "tick" may read variables in any order upon request (the actual order being determined by the VAX/VMS operating system), but no module is permitted to write on any cycle until all modules have completed their read-requests. Then all modules which are scheduled to write will write their output variables to common memory. In order to prevent overwrites, no two modules should have the same variable as an output. The foregoing events take place asynchronously as fast as the VMS operating system will allow, and the worst-case time determines the maximum emulation speed since the emulation runs on a single physical processor.*

The occurrence of the next clock "tick" is determined by the interactive display module. In single-step mode, the next "tick" (i.e., beginning of the next "read" cycle) occurs when the user issues a keyboard command. In variable-rate real-time mode, the user specifies the ratio of clock time to emulation time, and the elapsed time on the system clock determines the next tick.** In free-running mode, the next tick occurs immediately upon completion of the write cycle; the actual emulation speed may

---

* The total number of independent processes, the maximum number of common memory variables for each process, and the time for the operating system to serve a single process determine the actual maximum speed. The VAX/VMS operating system updates a process status every 10 msec.

** If the specified time has already elapsed, the next tick occurs immediately.

depend on the loading of the system by other users in this case.

The user also has several means with which to control when each module is scheduled to read or write. To start with, each module is assigned a basic scheduling interval when it is built into a VMS process as described in Section 4.3. A low-level module might be scheduled every tick, while a high-level module might be scheduled every 50th tick. The scheduling intervals have a significant impact on the running speed of the emulation. Furthermore, pre-defined variables are provided within each module to represent the effects of computing and communications delays. The compute-delay variable, which may be set depending on which computation a module executes or simulates on a given step, has the effect of delaying the writing of the outputs to common memory by the number of ticks which it specifies and of delaying subsequent reading from common memory; the communications-delay variable has the effect of delaying the output of a module by a fixed amount without affecting its scheduled reading rate.

Delays significantly affect the way in which modules are designed, because it cannot be assumed that current outputs will be read immediately by other modules. The intended recipient must acknowledge having received the output, or further output changes may be made without being seen by the receiver due to scheduling delays. A related observation is that inputs must be removed after they are acknowledged, or they will be re-executed. Unless every module is scheduled at every time-step, these

problems are not solved by common memory synchronization alone.

2.4 Simulation vs. Emulation Modules

Simulation modules, in contrast to emulation modules, represent the input/output relationships of segments of a controlled system, but not necessarily their internal structure. Usually, simulation modules represent non-digital (e.g., electromechanical, mechanical, chemical or thermal) parts of the controlled process, but they may also be used to represent digital subsystems or analog control system components. The HCSE module format is sufficiently general that simulation modules can be programmed in exactly the same way as emulation modules. However, the actual content of simulation modules will usually differ from that of emulation modules, and in order for emulation modules to be most effective, certain principles should be observed in composing the module.

Input and output variables of a simulation module should be chosen to correspond to signals which are readily identifiable and measurable in the actual physical control system under design. Typically, these can be classified as actuation or "control" signals, and sensory or "acknowledge" signals. The boundary between emulation and simulation may be drawn either between a central processor and device controller, or between the device controller and the physical process, depending on the desired level of detail. The most important principle for

selecting inputs and outputs is that parameters or variables from a simulation module should not be passed into an emulation module (and vice versa) unless there is to be a corresponding explicit measurement or control process in the actual physical implementation. In this way, errors in control system logic will be discovered much more readily. The conceptual model of plant behavior employed by the control system designer may or may not be consistent with the actual state equations governing parts of the plant, as incorporated in a simulation module, and the structure of the emulation/simulation should allow for this source of error if it is to be useful for control system design.

The input/output relations of a simulation module should represent as closely as possible the input/output relation of the physical subsystem being simulated. Often, such systems may be viewed as having a number of operating regimes (discrete states), where state transitions depend on the evolution of continuous variables (continuous state) within each regime. To simulate this situation, the (discrete) module states are placed in correspondence with the operating regimes. The module is scheduled to read and write at regular intervals. Each time the inputs are read, a variable-time-step integration procedure appropriate to the current state (i.e., regime) is called, and continuous variables are integrated forward one time-step. Finally, the conditions for switching between regimes are tested, the discrete state is updated, and the appropriate outputs are generated. Functions which return the current number of ticks,

the tick-spacing, and the system clock time are available to the user in order to properly synchronize the simulation. In the emulation/simulation of an event-driven control system, only the simulation modules will call these functions.

A simulation module may communicate global parameters to the other simulation modules (e.g., to determine when two independently-simulated physical objects come into contact) even though the variable is not sensed. Usually, the collection of all simulation modules constitutes a self-contained model of the environment, plant, or controlled system while the collection of all emulation modules constitutes a self-contained model of the feedback controller. Variables are passed between the distinct sets of simulation and emulation modules only if they correspond to explicit physical quantities which are measured or controlled. While all modules communicate information through one shared memory, the shared memory in the HCSE should be effectively partitioned (in a well-designed application) into a part corresponding to the emulated control system common memory, a separate part for the shared simulation variables (representing interaction between subsystems of the controlled plant), and interaction variables which represent sensor measurements and actuator command values.

3.0 SOFTWARE OVERVIEW AND GENERAL USAGE

Section 3.1 provides a user's overview of what software is contained in the emulation, while Section 3.2 describes in general terms how the software is used. The user should verify that the software described in this section is available on his VAX 11/780 VMS system before attempting to apply the operating sequence described in Section 4.

3.1 Software Overview

This section describes the principal software components of the HCSE, from the user's perspective. A complete set of filenames and libraries is listed in Section 8.1 of this manual. More detailed documentation of these programs may be found in the Programmer's Manual.

It is convenient to assume that the operational software is organized as a main directory which will be termed the HCSE_LIBRARY, along with three other libraries (of object files) termed the common memory library (CM_LIBRARY), the Praxis library (PRAXIS_LIBRARY) and the BP library (BP_LIBRARY)*, which is a utilities library. These libraries are accessed by the VMS linker through a linker options file in producing executable process images prior to running the emulation. Mostly, the

---

* BP library is a BBN-developed library supplied to NBS under a one-site license agreement. Need for this library may be eliminated in future versions of the HCSE.

executable files contained in the HCSE_LIBRARY are discussed in this manual, although references will be made to a few library routines which the user may find helpful in composing emulation/simulation modules.

Command files: Command files in the HCSE_LIBRARY include

```
FOREIGN.COM
BEGIN.COM
```

FOREIGN defines a set of VMS foreign commands which the user executes, as described in the next section, to compose, run, and evaluate an emulation. One of these foreign commands is the BEGIN command, which starts a named VMS process with privileges and options appropriate to the emulation; the BEGIN command invokes the BEGIN.COM file. The other foreign commands merely invoke executable process images which are described in the following paragraph.

Executable files: Executable files in the HCSE_LIBRARY include

```
PARSER.EXE
DICTION.EXE
BUILDER.EXE
DISPLAY.EXE
FORCEX.EXE
SMERGE.EXE
SIMLIST.EXE
SUMMARY.EXE
```

With the exception of DISPLAY, these images are executed (normally in the sequence shown) by the foreign commands PARSE,

DICT, BUILD, KILL, SMERGE, SIMLIST, and SUMMARY, respectively. DISPLAY is executed during the emulation using the foreign command DISPLAY. The Praxis compiler is invoked by the PRAXIS command, defined as a system global symbol. The list of directories searched by the Praxis compiler are defined as system logical names PRX$SYNOPSIS_0 through PRX$SYNOPSIS_4.

Object files: Object files obtained from the CM_LIBRARY and the PRAXIS_LIBRARY directories in the course of executing the foreign commands, command files, and the display program include

```
VAXDEF.OBJ
SHRMEM.OBJ
SHAREOUT.OBJ
RECREAD.OBJ
```

VAXDEF, SHRMEM, SHAREOUT, and RECREAD contain procedural primitives which implement common memory, and VAXRUNTIM.SPS* contains Praxis synopsis of VMS and Fortran library procedures used in the emulation.

---

* The file VAXRUNTIM.SPS is located in a synopsis subdirectory off the same main directory which holds the PRAXIS_LIBRARY.

3.2 Description Of Software Usage

Prior to attempting to run an emulation, as described in Section 4, the user should personally confirm that the directories, libraries, and files described in the previous subsection (3.1) exist on the system. In addition, if the tick-spacing is to be adjusted, the Programmer's Manual should be consulted.

After these initial steps, the process of coding, running, and analysis is achieved by a sequence of straightforward steps:

1. Source code for each module is entered by the user in finite-state machine (FSM) format using any available text editor.

2. The state machine code for each module is parsed to produce a corresponding Praxis module (PARSE command), which is then compiled (PRAXIS command) to produce an object module and a synopsis file.

3. Sets of Praxis object modules are linked together, with appropriate scheduling delays, to form concurrently executable VMS processes (BUILD command).*

4. A data dictionary for the full set of modules is produced in order to verify consistency of type declarations and variable names among modules (DICT command). This is useful for initial debugging purposes.

5. Each emulated process produced in step 3 is started up (BEGIN command). Lastly, the DISPLAY process is run (DISP command) and the emulation begins.

6. The user interacts with the display process at the terminal to monitor and control the progress of the emulation.

---

* Separate VMS processes may be used to emulate execution on different physical processors; logging and data analysis is on a per-process basis.

7. Upon completion, the user causes each process to exit gracefully (KILL command).

8. The user may list logging files (SIMLIST command), merge them (SMERGE command) and/or produce summary statistics (SUMMARY command).

A very simple example of this sequence is provided in Section 5. Of course, the above sequence of steps does not include the process of troubleshooting in cases where errors may arise; this is discussed in Section 6.

4.0 OPERATING SEQUENCE

The purpose of this section is to describe in detail each step in the use of the hierarchical control system emulation as outlined in Section 3.2. The most demanding step by far, is the formulation of state-machine code for each module of the emulation, which is necessarily the user's responsibility. The global issues of any particular application cannot be adequately addressd in this manual; for this purpose, the HCSE Applications Guide provides an example of significant complexity. This manual is restricted to specific issues that are generic to a broad class of applications, and only a very simple illustration is provided in Section 5 in order to clarify the format of user interactions at the terminal.

4.1 Entering State Machine Modules

General state machine concepts were summarized in Section 2.2. In the HCSE, a state machine module is entered by the user. A source file of type FSM uses the specific format described below, using the VMS EDT (EDIT) facility or any other editor.* The module format consists of an initial part with several individual subsections which are initialized by statements beginning with double-slashes (//). A second part follows.

---

* Although the emulation must be run on a DEC-supported terminal, or equivalent, source files may be entered from any terminal for which a text editor is available.

consisting of a collection of procedures written in Praxis which implement the detailed (or micro-state) calculations of the next-state and read-out maps. In parsing this source code, the initial section is translated into Praxis code. The parser also adds a suitable module template which provides read/write operations, and carries through the Praxis procedures from the second part of the source file unchanged. Among the advantages of this approach are that the source file has a simple standard format, that type-checking between modules can be preserved, and that the user is not obliged to repeat any tedious formatting details that are common to all modules of the emulation.

FSM Source File Format: The general format of an FSM module is shown in Table 1. The first part consists of lines which define types and declare input, output, state, and internal variable names in terms of these types, followed by a sequence of condition-action lines that implement rows of the state table. (The procedure for converting state-machine descriptions into state tables is summarizd in Section 8.2). Each line has a similiar structure

```
//identifier-token (space) declaration-token (space)
  type-token (space) | comment-text
```

where any one of the tokens may be blank (empty). The identifier-token may be:

(empty)	section already initialized by a previous identifer-token or the line is blank

TABLE 1. FSM MODULE FORMAT

```
//name  MODULENAME
//input  INPUTVARIABLE TYPE
  ...
//inparameter  INPUTPARAMETER TYPE
  ...
//internal  INTERNALVARIABLE TYPE
  ...
//outparameter  OUTPUTPARAMETER TYPE
  ...
//output  OUTPUTVARIABLE TYPE
  ...
//preprocess  STATEMENT
  ...
//postprocess  STATEMENT
  ...
//conditions  CONDITION1  ;  CONDITION2  ; ...
  ...
//actions  STATEMENT1  ; STATEMENT2  ; ...
[condition-action pairs]
//multimatch  STATEMENT
//nomatch  STATEMENT
//procedures
procedure PROCEDURE_1( )
  ...
end procedure {PROCEDURE_1}
  ...
[more procedures]
procedure  PROCEDURE_N( )
  ...
endprocedure {PROCEDURE_N}
[end of file]
```

Declaration Section

State-table Section

Procedures Section

name	module name (must agree with filename of FSM module)
include	type declaration file
type	type declaration
input	input variable declaration
inparameter	input parameter declaration
internal	internal variable declaration
state	state variable declaration
output	output variable declaration
outparameter	output parameter declaration
preprocess	preprocessed input variable declaration
postprocess	post processed output variable declaration
conditions	conditions for a state transition
actions	actions defining next state and output computations
multimatch	actions defining next state and output when multiple conditions are satisfied
nomatch	actions defining next state and output when no condition is satified
procedures	denotes beginning of procedures part of FSM module; all code after this line is strictly Praxis

One or more spaces delimit the remaining tokens. The declaration-token may be:

name_declaration	module name, if the identifier token is "name" file name, if the identifier is "include" user-defined type name, if the identifier is "type" variable name, if the identifier token is "input", "inparameter", "internal", "state", "output", or "outparameter"
statement_declaration	compound statement, if the indentifier token is "conditions", "actions", "multimatch", "nomatch", "preprocess", or "postprocess"
(empty)	if the identifier token is (empty) or "procedures"

Variable names, user-defined type names, and module names are strings of 29 or less characters that are valid as Praxis names (this excludes control characters, names that begin with nonalphabetic upper case symbols and Praxis reserved words). Type declaration files can have any file name acceptable to VAX/VMS systems. A compound statement consists of simple statements separated by semi-colons (up to a total of less than 132 characters per compound statement with no linefeeds, carriage returns, or other control characters). A simple statement may be either a logical expression (only when the identifier token is "conditions") or a valid Praxis statement (otherwise). A logical expression in this context is the same as a boolean expression in Praxis, with the important generalization that logical operations are considered to be well-defined between string variables of different lengths; thus the expressions

```
string 1 = "long-or-short-word"
string 1 <> "anything_else"
```

have meaning when string 1 and string 2 are (in general) different-sized arrays of characters.

The type-token is only recognized when the identifier-token is "type", "input", "inparameter", "internal", "state", "output", or "outparameter". In these cases the declaration-token (a name declaration) and the type-token must both be present. The type-token can take values:

char	Praxis character variable or parameter
integer_1	Praxis one byte integer variable or parameter
integer or integer_2	Praxis two byte integer variable or parameter
integer_4	Praxis four byte integer variable or parameter
real or real_4	Praxis four byte real variable or parameter
real_8	Praxis eight byte real variable or parameter
logical or logical_1	Praxis one byte logical variable or parameter
logical_2	Praxis two byte logical variable or parameter
logical_4	Praxis four byte logical variable or parameter

---

* This greatly enhances the clarity of the source code. The parser actually converts such expressions into valid Praxis functions in a later step.

boolean	Praxis boolean variable or parameter
pchar	Praxis character variable or parameter initialized $<NUL>
string	packed array of up to 31 characters (not a standard Praxis variable type)
or	
user-defined type	Praxis type declaration (can be one of three permissible data types: alias, array, or structure)

The first ten items in the list above are recognized by the parser as basic data types. The remaining items (except for user-defined types) are data types commonly used in FSM modules and their subsequent definitions are stored in a reference file named STANDARD.ISM located in HCSE_LIBRARY. The types declared in this file along with the basic types described above are automatically incorporated into FSM modules (by the parser) prior to the appearance of any type declaration lines in the source code.

In addition, the user has the capability to declare his own data types in terms of the predefined types presented above and/or other user-defined data types that were previously declared. Type definitions must pertain to either an alias, array, or structure data type construct and conform to the equivalent Praxis syntax (refer to the Praxis primer in Section 8.3). The three forms allowable are:

class: ALIAS

```
//type (alias_name) (type_name) [initially (init_value)]
```

where alias_name is the alias type being defined; type_name is either a basic type or a previously declared user-defined type which the alias type is being defined in terms of; and init_value is an optional initial value assigned to the type. (Initialization is only provided for alias types.)

class: ARRAY

```
//type (array_name) [packed] array (no_elements) of (type_name)
```

where array_name is the name of the array; packed is an optional parameter which indicates that the allocation of array elements in storage is packed or most efficient (unpacked is default); no_elements is the number of elements; and type_name is either a basic type or previously declared user-defined type.

class: STRUCTURE

```
//type (structure_name) structure
    (component_name) : (type_name)
    (component_name) : (type_name)
    (component_name) : (type_name)
  endstructure
```

where structure_name is the name of the structure; component_name is the name of an individual component; and type_name is a basic or user-defined type which can be used to define a component. The number of components in a structure definition is not restricted.

The user may place popular user-defined type declarations in a separate file and have FSM modules access these type definitions through the INCLUDE file statement. This eliminates the waste of repetitiously defining the same data types in a series of FSM modules. The //INCLUDE statement may appear anywhere within the initial portion of the FSM module; however, the type definition file must be attached (included) before any of its definitions appear in type or variable declaration statements in the FSM module. To ensure against a fatal compilation error, it is best to place the //include line prior to all type declarations in the FSM module. The type declaration file must reside on the same directory where the FSM modules are or directories whose logical names are FSM$INCLUDE_0 to FSM$INCLUDE_4.

A short example to clarify the use of user-defined types in a FSM module has been provided.

```
//type pinteger_2 integer_2 initially 0
//type longstring packed array(512) of pchar
//type db_mailbox structure
       mb_length : pinteger_2
       mb_seqno : pinteger_2
       mb_time : integer_4
       mb_data : longstring
  endstructure
```

Note that the types "pinteger_2" and "longstring" were defined before they could be used in the structure declaration. The two declarations may have been defined in another external file and attached to the above list of type declarations through the "//include (file_name)" statement. However, this approach is

only valid if the separate file is included before the structure type is declared. If the first two types were not declared before their appearance in the "db_mailbox" declaration, a compilation error would have resulted. The rest of the types used in the module are either basic types declared by the parser (integer_2 and integer_4) or types defined in the file STANDARD.ISM (pchar).

The comment delimiter in the first part of an FSM file is a vertical bar (|); all comments and the end-of-line are passed through as text by the parser whenever the comment delimiter is present. In the procedures part of the FSM file, the Praxis comment delimiter (double-slash) should be used after the line "//procedures" which inititates the second part of the FSM file; double-slash is no longer recognized by the parser as a line delimiter.

In summary, the structure of a FSM source file consists of a first part where lines beginning with a double-slash indicate the action to be performed. This initial part consists of a declaration section starting with a module-name line followed by lines which define all user-defined types, and then declare all variables used by the FSM module. Data type declarations are accomplished by attaching external type definition files to the FSM module and/or defining the data types themselves within the module. Following type definitions, inputs, input parameters, internal variables, state variables, outputs, and output parameters are declared. The current implementation of the

parser does not require that declaration lines appear in any specific order, but a user-defined data type must be defined before it is used in a variable declaration or in another data type declaration. In any case, it is advantageous to follow the order in which the type definition and variable declaration lines are specified in Table 1, for it greatly enhances clarity. The second section of the first part normally consists of optional preprocess statements, followed by as many condition-action sets as there are lines of the state table, followed by optional multimatch and no match actions, followed by optional postprocess statements, and ending with a line which inititates the procedures declarations. The second part of an FSM source file is written strictly in Praxis and normally consists of a set of procedures. The file ends with an end-of-line (EOL) character, which is generated automatically by the carriage-return key on a terminal; no other terminator should be used.

Composing Source Files: In composing FSM source files, a number of additional considerations can be used to produce compact modules and avoid programming errors. The following factors should be carefuly noted by the user.

(a) Upper and lower case: Upper and lower case variable names are not distinguished by Praxis, except in the values of constants or variables of type "char" and "pchar". For the convenience of the user, all alphabetic string values are converted to upper case so that the string "abcd" is regarded as equal to the string "ABCD".

(b) Predeclared variables: All variables and parameters employed in condition and action statements of an FSM source file must be pre-declared, with the exception of certain variables that are pre-declared by the parser and are available in all modules. The pre-declared variables which may be accessed by the user are:

Variable	Type	Explanation
curs	string	current state of the module
nexts	string	next state of the module
first_entry	boolean	flag set true only on the first cycle of a module
compute_delay	integer	simulated computing-delay in ticks
output_delay	integer	simulated output delay in ticks

The (implicit) identifier-token for these variables should be treated as "internal". The global variable name_curs, where "name" is the module name, is the global variable written to common memory corresponding to the local variable "curs".

The role of "curs" and "nexts" is obvious. By predeclaring the type of the state to always be "string", the user is free to choose names for the states that are convenient and meaningful in the context of his application. The number of possibilities is virtually unlimited. Using state names that have meaning greatly

simplifies programming and reduces programming errors.

By testing the variable "first_entry" in a condition statement, the user can predefine the initial state and internal parameters of a module in the corresponding action statement so that the module is self-initializing.

Output_delay is the number of ticks that the output is delayed each time a module is scheduled.* This emulates a communications delay; if the output delay is greater than the scheduling delay, the module may receive new information from common memory before its previous output has been written. Compute_delay is the number of ticks (in addition to the output delay) that the output is delayed by emulated computation; this variable may be defined within a procedure so that the delay depends on which state-transition is begin emulated. In contrast to the output-delay, a compute-delay will also delay the scheduled reading of variables from common memory whenever it is longer than the scheduled delay. In this case, the next read cycle occurs immediately upon expiration of the delayed write-cycle. The initial value of these variables is zero ticks (on every activation).

(c) Further discussion of the "state" variable: The values of state variables are both written into and read from common memory. Although this manual has emphasized that no module

---

* The scheduling interval is specified in the building step (see Section 4.3).

should input and output the same variable, there are moments when the internal activity of individual modules needs to be monitored to ensure that these modules are running correctly (or running at all). For example, in the current implementation of common memory, an FSM would go to sleep if its input variables do not change. This is a practical implementation since, in theory, inputs of a state machine determine its outputs; and if the inputs and outputs of a state machine do not change, it serves no purpose to continue cycling through the state transition table of that machine. However, with the utilization of internal variables in the HCSE, output values may change in response to internal activity such as an internal counter. The situation would result with the module being put to sleep, even though the states of the FSM are still in transition. This is a common occurrence in the HCSE and one of the motives behind designing the "state" variable. By declaring locally processed variables as "state", intermediate variables will be represented as input values which will subsequently cause module deactivation (that is, if state variables do change).*

(d) Local and global variables: Varables declared within procedures are local to these procedures. Variables declared by the identifer-token "internal" are local to the module produced by an FSM source file but are global with respect to the

---

* Another way to activate a sleeping FSM is to call the subroutine CM_WAKE_FSM. (see entry (k) for subroutine descriptions)

procedures within it. A variable declared as "inputs", "inparameters", "outputs" or "outparameters" are stored in common memory and hence are available to all other modules of the emulation. Variables declared as "state" is stored in common memory, but the variable is only intended for a single module to read and write. The predeclared variable "name_curs" is also stored in common memory at each write cycle and is available by this name to all other modules ("name" denotes the name_token of the FSM file).

(e) User-defined types: User-defined type declarations between different modules should be consistent. If individual modules define the same type name with different attributes, only the first definition passed into common memory will be used in the emulation. In addition, users are not permitted to modify the type definitions of basic types (see section 4.1). Any efforts to do so will be ignored. In either case, the appropriate error message will be output to the screen while the emulation continues to run unhindered.

(f) Variable-naming in FSM source files: A variable of the same name and type identified as an "output" in one FSM source file and "input" in another FSM source file will automatically be passed between them through common memory in the emulation. This is all that is required in order to transmit information between two modules through common memory. Otherwise, common memory is entirely transparent to the user. While this feature is very valuable, it also requires that a variable-naming convention be

adopted in advance so that different users will write modules which are compatible. Even slight differences in variable names (aside from case) will result in communication failure; the variables will be stored in common memory, but under different names; if two modules happen to use the same variable name for different purposes, an unintended communication may occur (although this will not happen if one of the variables is identified as "internal"). Only those variables identified as "inputs", "inparameters", or "states" will be retrieved from common memory on each scheduled read cycle, and only those variables identified as "states", "outputs", or "outparameters" will be written on each scheduled write cycle. Undeclared common memory variable names used in a module will result in fatal compilation or linking erors. The data dictionary capability (Section 6) is very valuable in identifying such discrepancies. There should never be any need to declare the same variable as both input and output of a module (except in the case of "state" variables). To do so will result in errors during compilation and linking. No two modules should declare the same variable as a state, output, or outparameter. This will not necessarily produce error messages but may result in undetected overwrite conflicts in common memory.

(g) Syntax of condition/action statements: In Praxis, the logical operator (=), used in condition statements and declaration sections, is distinguished from the equivalence operator (:=), used in action statements, by a preceding colon.

Condition statements are sequences of boolean expressions (separated by semicolons) while action statements are sequences of complete Praxis statements (also separated by semicolons); a condition statement line is not a valid line of a Praxis program.

(h) Procedure arguments: The arguments lists of procedures declared in an FSM source file should always be empty, as denoted by the empty argument list "( )" following the procedure name in the declaration and in all procedure calls within action statements. In view of the scoping rules (see (d)), procedure arguments should never be necessary. On rare occasions, it is useful to declare a variable as global to all procedures in an FSM source file without declaring it as an "internal" variable, e.g., a two-dimensional array. This can be achieved by including a valid Praxis declaration statement prior to the first procedure declaration in the FSM file.

(i) Line continuation: No line continuation character is provided in the first part of an FSM source file. Up to 132 characters are permitted on a source file line, however. In almost all cases, statements can be separated to appear on succeeding lines with the initial statement containing the identifier-token. In fact, a block of equivalent FSM statements needs only the first line to contain the identifier-token; all succeeding lines with blank identifier-tokens are automatically associated with the last identifier specified. However, identifier-tokens may appear in as many lines as the user wishes. Thus, a long state transition condition may appear on multiple

lines following a "conditions" token and/or multiple actions for a given condition can be stated on multiple lines following an "actions" token. However, an action line is always associated with the nearest preceding set of condition lines.

(j) Use of preprocess and postprocess variables: Preprocess statements are useful when different values of a particular function of the input and current state values may trigger different state transitions. Defining the value of a declared "internal" or "state" variable to equal this function, the values of the variable may be tested in successive condition statements -- e.g., state transitions may depend on the range of the sum of two real-valued input variables. Similarly, part of an output computation may be common to all state transitions and thus may be most readily placed in a "postprocess" statement. Intermediate parameters in this computation must be declared as internals, states, or through a procedure call.

(k) Default for multimatch and nomatch: If no "multimatch" or "nomatch" actions are provided and one of these conditions occurs, then no change will occur in the module state or parameters. If possible, the conditions statements of a module should be complete and mutually exclusive.

(l) External procedures: Procedures and functions from the libraries described in Section 3.2 are automatically retrieved when they are called from within a module. The following are of particular utility:

TST(string1,string2): a boolean function which is true when the value of string1 is equal to the value of string2.

MTH$SORT(real expression): A real function which returns the square root of its argument. Similarly SIN, COS and ATAN2 may be accessed.

MTH'$RANDOM(iseed): a real function which returns a random number approximately uniformly distributed between 0.0 and 1.0. This is called with a seed which is initialized with an integer and thereafter is called with the new seed returned by the function, i.e., the seed must be 'static'. This is useful in emulating random events.

CM_TICKSPACING (): a real function which returns the current tick-spacing in seconds (tickspacing is currently 0.1 seconds).

CM_GET_TIME(time,ticksize): a procedure which returns the emulated time in ticks (integer) and the tick-spacing (real). This is useful for wall-clock synchronization of the emulation and for emulating scheduled startup times.

CM_WAKE_FSM(n_fsm_name,h_fsm_name): A procedure whose function is to reactivate non-cycling FSM modules. As discussed earlier in (d), common memory was provided with a built-in safeguard that would force an fsm module to sleep (become non-functioning) if it continuously received unchanging input. A call to CM_WAKE_FSM

from a module will override the above disabling mechanism and force the module through its read and write cycles on a given tick. The parameters n_fsm_name and h_fsm_name (name and integer pointer of the fsm which is stored in common memory) are automatically assigned when an FSM module is parsed.

CM_LOGGING_ON(newflag,prevflag) A procedure which controls whether all variable transitions are logged in logging files. The parameter newflag is a boolean variable passed into the procedure and is true if system-wide variable logging is to be performed. Prevflag is also a boolean variable which holds the value of the previous logging state.

CM_LOG_VARIABLE(iptr,newflag,prevflag) A procedure which controls whether specific variables are logged. This procedure should be used only when system-wide logging is disabled. The parameter iptr is an integer handle which points to the location of a specific variable in common memory. If newflag is true, logging for the variable represented by iptr will be enabled. Prevflag indicates the previous logging state of that variable.

CM_DISABLE(fsm_name,fsm_handle) This procedure disables a specific FSM module so that it can no longer cycle through its state tables, even if input values change. The name of the FSM integer handle pointing to the location of the FSM name in common memory, is passed into the procedure.

CM_ENABLE(fsm_name,fsm_handle) This procedure enables a specific FSM module that has been disabled. However, this does not guarantee that a module will cycle through for it may be asleep (caused by unchanging input values). The name and integer handle are passed in as parameters.

Procedures for external file manipulation are also available through the Praxis textio library (see Section 8.3). Further library procedures and functions are documented in the Programmer's Manual.

4.2 Parsing State Machine Modules (PARSE, PRAXIS)

The result of the previous section is a file or collection of files in the user's current working directory with type FSM, for instance

MODNAME.FSM;version

The parser translates this into a Praxis source file in HCSE format; the command syntax is simply

PARSE MODNAME

The parser will automatically use the most current version of MODNAME.FSM, and will produce a file

MODNAME.PRX;version

in the current working directory with a version number one

greater than any previous file by this name. The parser checks for type declarations errors and outputs several different error messages. A common message is

```
Error: Non-existent (construct_component) type
(construct) name was: (name)
(construct_component) type was: (type)
```

which indicates that a type "name" was being defined in terms of a "type" that did not exist. "Construct" indicates whether the declaration was an alias, array, or structure definition and "construct_component" describes the particular component being defined (type, array element, or structure component). Remember that a type must be defined prior to its appearance in other subsequent declarations. If the user attempts to modify a basic type (see section 4.1), the following is outputted

```
Error: Illegal to redefine a built-in type
Type name was: (type)
Only first declaration will be used
```

with the "type" name returned to the user. If the user attempts to redefine another user-defined type previously declared, the parser outputs the error message

```
Error: Duplicate type declaration
Type name was: (type)
Only first declaration will be used
```

where the type name is again returned. If the parser cannot find a specified INCLUDE type declaration file, the message

```
Error in file name in an include statement
File name was: (file name)
```

will be printed on the terminal screen. Errors in FSM syntax will also cause an error message. If the parser is unable to recognize a token in the FSM code, the message

```
Unrecognized token in line - (line text)
Token was (token)
```

would be displayed on the user's terminal returning the actual line and token causing the error. All errors should be corrected in the source file and the parser should be run again before proceeding. If no error messages occur, the module should be compiled with the command

```
PRAXIS MODNAME
```

The praxis compiler will automatically use the most current version of MODNAME.PRX and produce files

```
MODNAME.OBJ;version
MODNAME.SPS;version
```

with appropriate version numbers in the current working directory. The second file is a Praxis synopsis file* used at build time to define the calling sequence of the Praxis funtion. Praxis error messages are described in Appendix E of the Praxis Language Reference Manual (see Section 1.3) and in the on-line file PRAXIS.DOC. Errors should be corrected in the FSM source file, but the PRX file produced by the Parser may be consulted for assistance in debugging if necessary. All errors should be

---

* See page 23.

traceable to the FSM source file, as there are no known Praxis errors that can be introduced by the parser.

4.3 Building Processes (DICT,BUILD)

The previous steps are performed for each of the modules in the user's application, resulting in an OBJ and SPS file for each module. At this stage, it is desirable to produce a data dictionary to assure that all of the modules use a consistent set of variable names. If the module names are MOD1, MOD2, ..., MODN, then the command

```
DICT MOD1, MOD2, ..., MODN
```

will display a data dictionary at the user's terminal. The standard keyboard commands CTRL/S, CTRL/Q can be used to suspend or continue the listing; this may be directed to a file by first issuing the system command

```
define/user SYS$OUTPUT DICT.LIS,*
```

which produces the file

```
DICT.LIS;version
```

in the user's directory. The dictionary begins with a listing of all user-defined types declared in the FSM modules, followed by a

---

* SYS$OUTPUT may be defined to any file the user wishes. The filename DICT.LIS was used in this example for convenience.

set of variable descriptions which are referred to as dictionary entries. The format of each dictionary entry is

```
(variablename)  (type)
     Written by : (modulename)
        Read by : (modulename)
     comments: (comments from variable declaration lines)
```

The dictionary program recognizes descrepancies between the type declarations of different modules. If an equivalent type name is given inconsistent definitions in two different modules, the message

```
Conflicting definitions for a user-defined type in
(modulename1) and (modulename2)
Type name:  (typename)
Latest definition will be given
```

will be outputted. In addition, the dictionary program also recognizes type conflicts in the variable declarations of different modules. The message

```
Type conflict (module name)
Changing type from (type1) to (type2) of (variable name)
```

will be printed prior to the dictionary listing for each error. In addition, the user should survey the list for spelling errors (which will generate separate entries), variables that have no reader or writer (which often indicates a failure to link two modules through common memory), and variables with multiple writers (which will lead to overwrite conflicts). Since comments are carried through from all modules, inconsistent comments for a variable may indicate that the same variable name or variable has

been unintentionally given different meanings in different modules. The data dictionary is very valuable for debugging. All FSM modules should be corrected, if any of these errors are noted, before proceeding.

The data dictionary only lists those variables which will be communicated through common memory (i.e., inputs, inparameters, and internals are not passed through common memory, nor are any variables defined within the procedures of a module. Any errors remaining in these variables must be determined at run-time or through the logging list (Section 4.7). Since only those variables in common memory can be monitored, the user must declare all variables that are to be monitored as "state" ,"outputs", or "outparameters" in the module where they are defined. Outputs of this type may appear legitimately in the data dictionary with no readers.

Under VMS, object modules must be linked in order to produce executable process image files. The versatility of the HCSE is greatly enhanced by the possibility of grouping the modules of an emulation into subsets which are linked and later executed as independent processes. The BUILD command constructs processes from groups of modules.

The user is free to decide on what basis the set of modules for an emulation should be partitioned into groups. Traditionally, all modules might be linked as a single group into one process. Another possibility is to link simulation and

emulation modules as separate processes. A third possibility is to group modules according to the emulated processor (computer) on which they are to be implemented, i.e., a separate process for each piece of hardware. Grouping by level in the hierarchy is another possibility.

Since different processes may run on different terminals, there is the possibility of grouping processes according to the terminal on which they are to be run. For instance, one terminal might run the emulation monitor display, while another might emulate (at actual speed) an operator's console, graphic display, or device interface. As described in the HCSE Applications Guide, a physical device with a serial interface (such as a robot) can actually be operated by the emulation while running its own process! The design of shared memory to support this sort of operation enormously increases the power of the HCSE over conventional simulation techniques.

Another consideration in grouping modules is that logging of variables is done on a "per-process" basis, so that the timing, computational burden, and common memory traffic within and between different processes can be readily monitored.

In order to link MOD1, ..., MODM, (M<N)*, in the preceding example into PROC1, the syntax of the BUILD command would be

```
BUILD PROC1 MOD1/interval1, MOD2/interval2, ...,
```

---

* The ellipsis (...) are only notational; this is not a feature of the BUILD command.

MODM/intervalm

where intervall ,...., intervalm are integer constants which denote the scheduling interval (in ticks) of each module. As described in Section 2.3, this defines the rate at which each module takes inputs from common memory. Unless the compute_delay and output_delay variables are used (Section 4.1), a module writes its outputs on the same cycle as it reads. Compute_delays longer than the scheduling interval simply cause a postponement of subsequent reads; in this case, synchronization of the reading-rates of various modules may not be maintained indefinitely during the emulation.

The scheduling intervals of various modules may significantly affect the run-time efficiency of the emulation because they strongly affect the rate and volume of data transfer in and out of common memory. Thus, modules should be scheduled at the lowest rate consistent with the task or subtask they implement, with the required degree of responsiveness to errors occuring in other modules, and with the emulated computing-time requirements.

In the above example, the BUILD command generates a Praxis file

PROC1.PRX;version

in the user's current working directory. The builder also constructs a linker options file

PROC1.OPT;version

with the appropriate version number. The process-building step is completed by compiling and linking the above module:

```
PRAXIS PROC1
LINK PROC1/OPT
```

which produces the executable process image

PROC1.EXE

4.4 Running The HCSE

An emulation consisting of K process files PROC1, ..., PROCK is run by starting each process and finally starting the DISPLAY process, which is designed as a special process module that acts like a probe for the ongoing emulation and provides a user interface.* The user issues commands

```
BEGIN PROC1
  ...
BEGIN PROCK
```

to start all of the emulation processes. In a hierarchical control system emulation, only the top module will normally proceed, while the others will wait in an "idle" state until receiving their first commands. The BEGIN command reserves sufficient system resources for the emulation. Next, the display

---

* The emulation can be run in batch mode by omitting the display.

process is run be issuing the command

DISP or DISPLAY

This process sends output directly to the user's terminal. The first request is

"Hit any key to start display".

However, the "E" and "D" keys are reserved for functions (described later) which require the emulation to be running before they can have an effect. These keys should not be pressed to initiate the display. Doing so may stall the DISPLAY process. After the initial key is struck, the emulated time is displayed and the user may proceed by typing "H" or "?" which will display the text

"Your options are:

E - enter a new variable for display
D - delete a variable from the display
M - move display window up or down
up arrow - move display window up
down arrow - move display window down
right arrow - move display window right
left arrow - move display window left
C - change speed of emulation
S - single step simulation
G - go, resume continuous operation
Q - quit (exit program)
L - log a snap shot file
H or ? - this text

Hit any key to continue."

These commands have the following effects.

Enter (E): The display process then issues the request

"Variable name:"

to which the user responds with the name of a variable in common memory whose value is to be monitored. The response to this command may include asterisks as wild-card characters -- all variables which match the remaining characters of the response will be displayed. This display is then maintained in real time until the user makes another request. The display is a list of entries of the form

VARIABLETYPE VARIABLENAME VALUE

where the user can observe VALUE to change dynamically as the emulation proceeds.

Delete (D): The display process issues the request

"Variable to be deleted:"

and the remaining features of this command are like the "enter" command; the specified variables are deleted from the display list.

Move Display Window (M): This command allows scrolling of the display window and should be used when the display list exceeds the vertical boundaries of the terminal screen. (The current dimensions of the display screen are 23 lines down and 80 columns across.) After the "M" key is pressed, the display process prompts

"How many lines?"

whereby the user's response may be any non-zero positive or negative integer. A positive number will scroll downwards while a negative number scrolls up. The move command should only be issued after an "enter variable (E)" or "delete variable (D)" command for only these two functions are capable of outputting a listing large enough for scrolling to be required. In any case, the window will not scroll pass the top nor the bottom of any list. In addition to the move command, the four arrow keys on the terminal keyboard will shift or scroll the window one line or column in their respective directions. Shifting sideways may be required when information contained on one line exceeds 80

columns. Like the move command, the four arrow keys will not scroll and shift pass the edges of the list.

Change Speed (C): This command allows the user to adjust the ratio of real time to emulated time and thus to synchronize the emulation to clock time (the actual clock is the VAX internal time standard). In response to this command, the display process issues the request

"Enter speed ratio (real time to emulated):"

The user responds with a real number that is the ratio of real time to emulated time. Note that a larger number makes the emulation run more slowly. At some speed, depending on the size of the example (and in a time-sharing environment, the current computational demands of other users), the emulation necessarily becomes compute-bound. In this case, the emulation proceeds at its maximum possible speed whenever this is less than the specified speed. Thus a response of zero yields the maximum compute-bound speed of emulation; this is also the default mode of emulation.

If the user wishes to perform detailed timing studies which require synchronization with external devices, it is recommended that he operate VMS as a single user. However, for casual monitoring purposed, the user will also find the speed command useful for slowing down the emulation in order to watch critical transition sequences during debugging. The speed and the set of displayed variables can be changed at any time during emulation.

Since the current value of the tick-spacing is set at 0.1 seconds, a speed ratio of 1.0 will typically result in values changing on the display at a rate of up to 10 values per second, which is often too fast to watch. Thus, ratios of 10.0 to 100.0 are generally best for viewing. Since all timing and scheduling delays are normally specified in "ticks", it is recommended that the user estimate these values so that a speed ratio of 1.0 gives actual real-time operation.

Single Step (S): Issuing the "S" command causes the emulation to stop. The emulation proceeds one tick every time the "S" key is depressed. Depressing a command key causes the specified command to be executed. This command is useful for single-stepping through critical sections of the emulation to trace individual transitions. Continuous operation is resumed with "G".

Go (G): This command causes the emulation to resume running at current speed, displaying all currently entered variables. No prompt is given to the user.

Log a Snapshot (L): This command creates a file which contains the current contents of common memory, in response to its request

"Enter snap shot file name:"

The user specifies a valid VMS filename, such as

STATUS.LOG

and this snapshot file is deposited in the current working directory.

The snapshot log file may be later displayed on the user's terminal with the SIMLIST command (Section 4.6). It contains a record of the emulated time, the VMS system time, and a list of the names and values of all variables in common memory at that time. The emulation continues automatically following a snapshot, as with the E, C, or G commands.

Quit (Q): This command causes a graceful exit of the display program, and a return to the VMS monitor. However, the operation of the emulation and data logging is actually continued. The acknowledgement of this command is

"Exiting"

In order to terminate the emulation, for this example of K processes, the user executes the commands

KILL PROC1
...
KILL PROCK

which gracefully terminates each of the emulation processes. After each command, an exithandler* acknowledges

"Exiting"

---

* See Programmer's Manual (SIMLEAVE).

when it closes the logging file for that process.* As the process exits, the exit handler appends VMS process statistics such as elapsed time, CPU time, buffered and direct I/O and page faults, in addition to the total number of common memory reads and writes that the process made in the course of the emulation. Then it deposits the logging file in the current working directory. The above sequence of commands would create logging files

PROC1.LOG;version
...
PROCK.LOG;version

These may be analyzed following the emulation as described in the next step. At the exit of the last emulation process, VMS reclaims the storage which was allocated to common memory.

* The "Exiting" acknowledgment may not appear on the terminal screen directly after a KILL command has been issued. If this should happen, continue depressing the <RETURN> key successively until the proper "Exiting" acknowledgment occurs.

4.5 Log File Output And Summary Statistics

Following an emulation, the current working directory will contain log files for each process and for each snapshot that was recorded during the emulation. Any log file (e.g., PROCESS.LOG or STATUS.LOG) may be listed on the user's terminal by a command

SIMLIST PROCESS

or

SIMLIST STATUS

Each record in the log file begins with a single character identifier which specifies the type of information that the record contains. SIMLIST will retrieve each individual record from the logging file and reformat it into a more intelligle form. The first two lines of a process log file contain the process name (IMAGE_NAME) and the tick_spacing of the emulation. User-defined type declarations usually appear next in the appropriate Praxis syntax. To accomodate the construct of structure type declarations, multiple records are used to store the various components, thereby, without modifying the structure's form. Variable transition records are listed next with a three column format:

emulated time name value

In this section, only changes of variables are logged, hence the "time" column only contains times when values of variables have

changed: this information is sufficient to reconstruct a complete picture of all common memory values associated with the process at any time.* In the case of a snap shot, the "time" column of a snapshot file contains the last time that each variable in common memory was changed prior to the snapshot; the "value" column contains the value at exactly the time the snapshot was taken (which is the same as the value following the most recent change, of course). A summary of the process statistics is listed at the end of the logging file. The procedure for creating a listing file that can be printed is similar to that for the DICT command (Section 4.3), e.g.,**

define/user FOR006 PROCESS.LIS.

Summary statistics for a file PROCESS.LOG are generated by the command

SUMMARY PROCESS

which produces a listing on the user's terminal which initially contains a list of all user-defined types declared in a process. The next section holds, for each distinct variable in the log

---

* From the description of logging files, the user may receive the impression that there is a distinct separation of type and variable records in a logging file. However, programs which are called in the course of an emulation may also declare types and thereby, insert a seemingly misplaced type definition amidst variable transition records in a logging file. However, this will have no effect on the SIMLIST program for it reads through the logging file twice and separates types and variables before listing them.

** FOR006 is the system logical name for SYS$OUTPUT.

file, its final value, number of transitions, the set of values it assumed (maximum and minimum values, for real and integer variables), and the percentage of total emulated time spent in each value (omitted for real or integer variable). The procedure for creating listing file that can be printed is the same as for the DICT command (Section 4.3).

Several process log files may be merged prior to issuing the SIMLIST or SUMMARY commands. If PROC1.LOG, ... ,PROCK.LOG are the LOG files of K processes, then the command

SMERGE PROC1,PROC2

will create the file MERGED.LOG representing two processes. The listing produced by SIMLIST will indicate by "IMAGENAME" when each process was begun (by the BEGIN command) and by a set of process statistics when it was KILLed. The entry and exit of the display module have no effect on the listing file.

5.0 SAMPLE DIALOG

This section contains a very simple two-module example to illustrate the complete sequence of commands for running the emulation. Module COUNT1.FSM resets to -10 and thereafter increments (like a simulated integrator counter) on each tick until it is reset again. Since the variable "count" does not represent an actual state of the state machine, but rather an internal or intermediate state, count was declared to be a "state" variable. Module COUNT2.FSM observes the state of COUNT1

through common memory and issues a "RESET" command to COUNT1 when its count reaches +10 (like a reset controller). These are to be combined in a single emulation/simulation process UPDOWN; this will only work correctly if the two modules successively communicate via commom memory. The source codes for COUNT1 and COUNT2 are:

```
//name count1
//input command string
//output count integer
//conditions command = "up"
//actions count := count + 1
//conditions command = "RESET"
//actions COUNT := -10
//multimatch nexts := "MULTI"
//nomatch nexts := "NOMATCH"
//postprocess CM_WAKE_FSM(n_fsm_name, h_fsm_name)
//procedures


//name count2
//output command string
//input count integer
//conditions curs = "<NUL>"
//actions nexts := "RUNNING"
//conditions curs = "RUNNING";count<10
//actions command := "up"
//conditions curs = "RUNNING";count>=10
//actions command := "RESET"
//postprocess CM_WAKE_FSM(n_fsm_name, h_fsm_name)
```

FIG. 4. MODULES COUNT1 AND COUNT2

Note that COUNT1 is unusual because its current state is never read from memory and because it does not initialize the value of the "count" variable; command values are both upper and lower case. While neither module requires procedures, COUNT1 contains a "//procedures" statement while COUNT2 does not.

```
module COUNT1
export COUNT1
use mathlib, textio, vax_run_time, shared_memory, shareout, vaxdef
function COUNT1() returns compute_delay : integer initially 0

declare

//-              SYMBOLIC NAMES FOR TYPES
   INTEGER_1 is 8 bit integer
   INTEGER_2 is 16 bit integer
   INTEGER_4 is 32 bit integer
   REAL_4 is real
   REAL_8 is long_real
   LOGICAL_1 is 8 bit logical
   LOGICAL_2 is 16 bit logical
   LOGICAL_4 is 32 bit logical

//             HANDLES FOR PREDEFINED TYPES
   h_CHAR = 1
   h_INTEGER_1 = 2
   h_INTEGER_2 = 3
   h_INTEGER_4 = 4
   h_REAL_4 = 5
   h_REAL_8 = 6
   h_LOGICAL_1 = 7
   h_LOGICAL_2 = 8
   h_LOGICAL_4 = 9
   h_BOOLEAN = 10
   h_INTEGER = 11
   h_REAL = 12
   h_LOGICAL = 13
   h_LONG REAL = 14

//                       USER-DEFINED TYPES
   PCHAR is CHAR initially $<NUL>
   STRING is packed array[1..32] of PCHAR

//             NAMES FOR USER-DEFINED TYPES
   n_PCHAR : static string initially "PCHAR"
   n_STRING : static string initially "STRING"

//             HANDLES FOR USER-DEFINED TYPES
   h_PCHAR : static integer
   h_STRING : static integer

//             INPUT VARIABLES
   COMMAND : static STRING
   n_COMMAND : static string initially "COMMAND"
   h_COMMAND : static integer

//             OUTPUT VARIABLES
   COUNT : static INTEGER
   n_COUNT : static string initially "COUNT"
   h_COUNT : static integer

//             INTERNAL VARIABLES
   n_fsm_name : static string initially "COUNT1"
   h_fsm_name : static integer initially 0
   curs : static string
   n_curs : static string initially "COUNT1_CURS"
   h_curs : static integer
   nexts : static string
   first_entry : static boolean initially true


   time : static integer
   output_delay, pairnumber, pairselect : integer initially 0

enddeclare

//       GET HANDLES ON ALL TYPES AND IN/OUT VARIABLES
   if first_entry do
     h_PCHAR := CM_OPEN_ALIAS_TYPE(n_PCHAR, h_CHAR)
     h_STRING := CM_OPEN_ARRAY_TYPE(n_STRING, h_PCHAR, 32, true)
     h_curs := CM_OPEN_VARIABLE(n_curs, h_string, n_fsm_name, h_fsm_name, SI)
     h_COMMAND := CM_OPEN_VARIABLE(n_COMMAND, h_STRING,
     n_fsm_name, h_fsm_name, SI)
     h_COUNT := CM_OPEN_VARIABLE(n_COUNT, h_INTEGER,
     n_fsm_name, h_fsm_name, SO)
   endif
```

Fig. 5

```
COMMAND : STRING
    Written by : COUNT2
       Read by : COUNT1
       comments:

COUNT : INTEGER
    Written by : COUNT1
       Read by : COUNT2
       comments:
```

FIG. 6. PARTIAL DICTIONARY LISTING FOR COUNT1 AND COUNT2

The command

PARSE COUNT1

produces the file COUNT1.PRX shown in Figure 5. The details of this code are unimportant here, except to note that the comment "do the counting" from the condition/action section of COUNT1.FSM has been correctly carried through to the PRAXIS code to faciliate debugging.

The command

```
PRAXIS COUNT1
```

then produces the files

```
COUNT1.OBJ
COUNT1.SPS
```

The same procedure is repeated for COUNT2.FSM.

These two modules could be built either into separate processes or into a single process; the latter option was selected. The command

```
DICT COUNT1,COUNT2
```

produces the record in Figure 6 at the user's terminal. This indicates that the variable "count" of type "integer" is passed for COUNT1 to COUNT2 and that the variable "command" of type "string" is passed from COUNT2 to COUNT1. Since this does not indicate any errors, proceed with the command

```
BUILD UPDOWN COUNT1,COUNT2
```

which produces files

```
UPDOWN.PRX
UPDOWN.OPT
```

in the current working directory. No scheduling delays were used in this example. For completeness, the file UPDOWN.PRX is shown in Figure 7. The statements

```
main module UPDOWN
use shareout, shared_memory
use COUNT2
use COUNT1
declare
   COUNT2_count : integer initially 0
   COUNT2_delay = 0
   COUNT1_count : integer initially 0
   COUNT1_delay = 0
enddeclare
repeat
   if COUNT2_count = 0 do
      COUNT2_count := MAX(COUNT2().COUNT2_delay)
   otherwise COUNT2_count *= -1; endif
   if COUNT1_count = 0 do
      COUNT1_count := MAX(COUNT1(),COUNT1_delay)
   otherwise COUNT1_count *= -1; endif
CM_DUMP_OUTPUTS()
until false
endmodule {X}
```

FIG. 7. UPDOWN.PRX

```
use COUNT2
use COUNT1
```

in this file call on the Praxis compiler to use the modules COUNT1.SPS, COUNT2.SPS produced in the previous step. To compile and link this combined process, the user types

```
PRAXIS UPDOWN
LINK UPDOWN/OPT
```

which produces the file UPDOWN.EXE. Now to run the emulation, one types

```
BEGIN UPDOWN
DISPLAY
```

and the display question

"Hit any key to start display"

appears on the screen, along with the emulated time, as explained in Section 4.5. Begin with any key other than E or D. After the process has begun, enter

E

the user is asked

"Variable name:"

And since this is a small example, the response

*

will show all the variables in common memory. This includes

```
COUNT1_CURS: (value)
COUNT2_CURS: (value)
count: (value)
command: (value)
```

The value of COUNT1_CURS remains "NOMATCH"; COUNT2_CURS remains "RUNNING". The value of count is seen to increment rapidly from 0 to 11, and thereafter it is periodically reset to -10 whenever it has counted up to 11. The value of the variable command remains as "up", and as the count reaches 11, it goes to "RESET". Instantaneously after the count goes to -10, command is reset to "up".

One might wish to record a snapshot by typing

L

and supplying a file name SNAP.LOG in response to the query

"Enter snap shot file name"

or to go into single-step mode with

S

in order to collect a set of snapshots at successive times to

examine why the variable count achieves the (perhaps unexpected) value of eleven.

Curiousity exhausted, the user may wish to type

Q

and end the emulation. This elicits the acknowledgement

"Exiting"

as DISPLAY passes away. But the emulation actually continues until the user types

KILL UPDOWN

upon which the system replies

"Exiting"

and returns the VMS prompt ($). Recall that the "Exiting" reply may not appear on the terminal screen immediately after the KILL command is issued. If this should happen, simply press the <RETURN> key several times and the proper "Exiting" response will surface.

To see the snapshot and logging file, one types

SIMLIST SNAP

and/or

SIMLIST UPDOWN

The relatively long listing produced by the second command can be suspended and continued with CTRL/S and CTRL/Q at the user's terminal. The first and last page of this listing is shown in Figure 8.

From this, the following observations may be quickly noted:

(1) On the first cycle COUNT1_CURS goes to "NOMATCH" because the initial value of command is "<NUL>". Thus, on the first cycle, none of the conditions was true and module COUNT1 goes to the next state "NOMATCH" as specified. No subsequent actions change the_state from this value. Module COUNT2 self-initializes to state "RUNNING".

```
  User-defined types in order of declaration
   PCHAR is CHAR
   STRING is packed array[1..32] of PCHAR

Tick Spacing: 0.100000
0:0:0.00    COUNT1_CURS  NOMATCH
0:0:0.00    COUNT2_CURS  RUNNING
0:0:0.10    COMMAND  up
0:0:0.20    COUNT  1
0:0:0.30    COUNT  2
0:0:0.40    COUNT  3
0:0:0.50    COUNT  4
0:0:0.60    COUNT  5
0:0:0.70    COUNT  6
0:0:0.80    COUNT  7
0:0:0.90    COUNT  8
0:0:1.00    COUNT  9
0:0:1.10    COUNT  10
0:0:1.20    COUNT  11
0:0:1.20    COMMAND  RESET
0:0:1.30    COUNT  -10
0:0:1.40    COMMAND  up
0:0:1.50    COUNT  -9
0:0:1.60    COUNT  -8
0:0:1.70    COUNT  -7
0:0:1.80    COUNT  -6
0:0:1.90    COUNT  -5
0:0:2.00    COUNT  -4
0:0:2.10    COUNT  -3
0:0:2.20    COUNT  -2
0:0:2.30    COUNT  -1
0:0:2.40    COUNT  0
0:0:2.50    COUNT  1
0:0:2.60    COUNT  2
0:0:2.70    COUNT  3
0:0:2.80    COUNT  4
0:0:2.90    COUNT  5
0:0:3.00    COUNT  6
0:0:3.10    COUNT  7
0:0:3.20    COUNT  8
0:0:3.30    COUNT  9
0:0:3.40    COUNT  10
0:0:3.50    COUNT  11
0:0:3.50    COMMAND  RESET
0:0:3.60    COUNT  -10
0:0:3.70    COMMAND  up

     STATISTICS
  Terminated at:   0:0:26.30
  Total common memory reads  1052
  Total common memory writes  1052
  Elapsed Time  00:00:21.83
  CPU Time (10 msec units)  375
  Buffered I/O  3
  Direct I/O  2
  Page faults  150
```

(2) At 1.2 seconds, COUNT2 has detected that count is 10 and changed the command to "RESET", but on the same cycle COUNT1 has deposited 11 in common memory. On the following cycle, count is reset to -10 in module COUNT1, as desired, but module COUNT2 still reads the previous value, count=11, so that its output is unchanged. At 1.3 seconds, module COUNT1 resets again to -10, since this is no change from its previous value, this unintended situation is <u>not</u> recorded in the log file. But at this same time, COUNT2 detects the reset value of count, and switches its command to "up".

(3) Counting proceeds as intended, until the situation of the previous step is repeated.

(4) At the end of the log file, the total number of emulated common memory "read" and "write" operations is tabulated (these are almost equal, since each module reads and writes a single variable in common memory). The last 8 lines of the log file are VMS statistics which allow the user to gauge the computational requirements of the emulation itself. The fact that the elapsed time (7.3 sec) was close to the emulated final time (7.2 sec.) is a coincidence; the example was run on a very heavily-loaded system. The actual CPU time consumed by the emulation was approximately 115 x 10 msec. or 1.15 seconds; thus on a single-user basis, this example could run at approximately 10 times emulated speed. A large number of page faults in the log file is an indication that the emulation itself is quite large,

and/or that more system resources should be allocated to the emulation.

Finally, the command

SUMMARY UPDOWN

will produce the summary statistics shown in Figure 9.

```
          IMAGE NAME: DRA1:[MHS.BUFFER2]UPDOWN.EXE;36
     Terminated at:   0:0:7.30
     Total common memory reads  362
     Total common memory writes  284
     Elapsed Time  00:00:02.54
     CPU Time (10 msec units)  115
     Buffered I/O  3
     Direct I/O  0
     Page faults  134

    SUMMARY OF VARIABLE STATISTICS
    NOTE: VALUES GIVEN ARE LAST VALUES
    WARNING: TRANSITIONS MAY BE AFFECTED BY INITIAL VALUES

  COUNT1_CURS  NOMATCH (TRANSITIONS: 1)
  Total transitions for that variable: 1

  COUNT2_CURS  RUNNING (TRANSITIONS: 1)
  Total transitions for that variable: 1

  COMMAND  up (TRANSITIONS: 7)
           List of values
           Duration     %    Value
           0:0:0.60     8    RESET
           0:0:6.60     90   up
  Total transitions for that variable: 7

  COUNT  2 (MIN: -10  MAX: 11  TRANSITIONS: 68)
  Total transitions for that variable: 68

  Total number of variable transitions: 77
```

This shows, among other facts, that there were 68 changes in the integer variable "count", which ranged from -10 to 11, and that the variable "COMMAND" spent about 8_ of the emulated time at the value "RESET".

This example demonstrates approximately correct behavior, even though the synchronization between modules is not exactly what the reader may initially have expected. This illustrates

that even though each module is scheduled on every tick, and even though common memory is functioning precisely as described, the communications between modules cannot be assumed to be automatic. In more complex cases, several cycles of "handshaking" may be required in order to achieve an intended interaction.

The reader is invited to attempt to modify this example so that "count" does not reach the value 11 and does not repeat the value -10 twice, and so that the state of module COUNT1 self-initializes correctly. As a further test, the effects of building other versions of UPDOWN with various independently specified scheduling intervals for COUNT1 and COUNT2 should be observed. This type of requirement is more typical of general-purpose asynchronous emulations.

6.0 ERROR MESSAGES AND DEBUGGING

A step-by-step process for writing and running an emulation has been described in Section 4. Possible sources of error were indicated in several of the steps. As a rule, all errors discovered at a given step should be corrected (usually by modifying FSM module source code and repeating the prior steps) before proceeding with subsequent steps in the process. Usually, error messages generated by the emulation modules can be corrected directly by reference to the source files. A complete set of error messages produced by the emulation code is provided in the Programmer's Manual. The emulation itself provides two of the most valuable debugging tools: the DICT and SIMLIST commands. As indicated in Section 4, DICT allows the user to rapidly check that modules are consistent in their naming conventions and that minor typographical errors have not occurred in variable names. These errors may otherwise go undetected: the emulation will run, but the correct connection will not be made through common memory. The logging file produced by SIMLIST is an excellent aid in determining logical errors in communicating between modules, as illustrated by the example of Section 5. Generally, receipt of a command should be acknowledged, which should result in the command and acknowledgement being reset. This logic must occur within the modules if communication is to be robust in the presence of scheduling delays. The logging file produced by SIMLIST is the only diagnostic which contains the actual values assumed by

variables, as these are not readily determined until run-time. Often an error can be detected by reference to the summary listing, which may reveal that a variable did not take on one of its expected values during a run, or did rapidly transition in and out of an unexpected value.

In addition to error messages produced by the emulation code, other error messages can be produced by the Praxis compiler, the linker, and the VMS operating system. The user should ascertain which program has given rise to an error message from the syntax of the message. If the VMS message "Unrecognized command" is issued on response to the PARSE command, the user should re-execute the foreign command file by typing

@HCSE_LIBRARY:FOREIGN

This is always necessary when the user logs in and should be in the user's login command file. If the executable images of the operational software programs in the HCSE_LIBRARY are missing or lost, the entire set can be regenerated provided that the various object and linker options files are still in existence.

VMS and the HCSE always select the most current version number of a file for processing, and also automatically seeks the proper file type. If all steps of a revision are not completed, then the (erroneous) previous file version will be used if a later step is attempted; usually this results in a fatal error. In addition, it is recommended that the user adopt the suggested file type specifications of this manual in order to avoid

retrieving an incorrect file.

Error messages generated by the Praxis compiler are listed in the Praxis Language Reference Manual. These are most likely to originate from the "//procedures" segment of an FSM source file. Another possible source of compiler errors may arise if the user makes changes directly to the Praxis source code produced by the FSM parser. Occasionally, review of this code may be useful during debugging, but it is recommended that corrections be entered through the FSM source file rather than in the PRX file produced by the parser.

7.0 PERFORMANCE CAPABILITIES AND LIMITATIONS

The purpose of this section is to assist the prospective user in evaluating the HCSE concept in comparison to more traditional simulation methods, and to provide indication of some specific techniques which have been developed in the HCSE Applications Guide.

7.1 Limitations

The HCSE is machine-dependent and is not readily transportable to machines other than the Digital Equipment Corporation VAX with the VMS operating system. It is also based on the Praxis language, which is not yet in widespread use. While the HCSE concept as described herein can be implemented on other machines, using other programming languages, its performance capabilities and limitations would depend significantly on the features of the particular machine and language chosen. Clearly, a moderately large machine, with substantial disk storage, and a relatively powerful language which allows procedures to be separately compiled, are prerequisites in obtaining adequate performance.

Since Praxis is a compiled language, all modules of the emulation must be compiled prior to running the emulation. Thus, the procedural code, operators, and local variables may not be changed at run-time. In some respects, this is less convenient than an interpretive language (such as FORTH or LISP) would be.

The feasibility and relative efficiency of implementing the HCSE concept in an interpretive language is not clear at this time.

The HCSE runs on a single serial processor, and this places a distinct limitation on the size and/or speed of emulation which can be run at real time speeds. VMS updates process event flags every 10 msec., and this places an upper limit on execution speed. While there is nothing in the concept of a modular sensory-interactive hierarchical control system that would prevent multiprocessor implementation, the practical implementation of shared memory becomes increasingly difficult as the number of concurrent processes and processors becomes large.

The HCSE may require more expertise and careful preparation by the user than conventional simulation. Conventional simulation would represent the input-output relations of modules, but not the structure of internal software of the actual system or its run-time properties.

The use of a shared memory for interprocess communication, and the state-machine format of the modules give rise to certain rather subtle limitations of the HCSE. For instance, it is the total number of variables read and written on each cycle which most affects the run-time efficiency of the emulation. The state-machine structure of the module is always a limitation compared to the possibility of simply writing a program module in Praxis -- in an asynchronous application where the relative reading and writing rates of different modules vary, the length

of the necessary send/acknowledge sequences can become quite long, especially when several modules must be synchronized or when resource allocation is required. Operations such as counting or queuing, which are not computable by finite-state machines, must be embedded in local procedures,

7.2 Performance Capabilities

In the realm of control system design, the use of an interactive real-time emulation/simulation is new, and the HCSE represents one of the first systems of its kind. By contrast to conventional batch-process simulation tools, the HCSE has significant advantages for industrial applications.

In many ways, the emulation of timing is easier than simulation of timing, because complex temporal interactions among computational processes are difficult to describe in simulation programs. Of course, the common memory synchronization employed in the HCSE is not as general or as complex as scheduling processes employed in computer operating systems and communication networks, but it is still very complex.

The possibility to actually experience, stop, examine, and even modify the course of an emulation greatly facilitates debugging of a control system, and the thoroughness of the debugging is greatly enhanced in an emulation as compared to a simulation.

As indicated in the introduction, emulation permits a quantitative evaluation of the adequacy of computational resources and alternative allocations of multiprocessing resources, which would otherwise be very difficult to determine.

The capability to vary the rate of the emulation is extremely useful for purposes of interactive display. Note that graphics display (although not described in this manual) can be run concurrently with the monitor display as separate processes. The rate of emulation can be varied from single-step to slow or fast wall-clock synchronization, to the maximum rate compatible with the current utilization of machine resources. The emulation runs in a time-sharing environment.

Perhaps the most powerful feature of the emulation is the capability to run at exactly real-time speed and to interface with actual subsystem components. By first doing a purely software emulation/simulation and then gradually replacing simulation modules by actual pieces of equipment, the complete control system can be "brought up" on a piece-by-piece basis. At this point, the emulated software can (again, possibly on a subsystem basis) be replaced by the dedicated target software. The potential economic benefits of this procedure are very substantial: reducing plant startup time and permitting subsystem operation prior to the arrival of all plant equipment. During start-up, the full cost of capital must normally be paid while there is no production; in a high-interest rate economy, reducing startup time can lead to substantial savings. A related

use of the HCSE is to provide back-up during failures of control system components without the expense of redundant computer capacity that is often conventionally employed to provide duplicate back-up capability. Although the HCSE would not normally have the speed of a dedicated control system, it could provide a reduced level of function in emergency situations.

The easiest way to run the emulation with a synchronous external device is to set the speed ratio to 1.0 on the display and to use a clock output derived from the VAX internal time standard to clock the external device. Since the clock time at which the emulation is begun is typically not known in advance, an operation which is to begin at a certain time-of-day should be synchronized by use of VMS system services. If a synchronous external device does not permit an external clock signal, an independent process may be written to synchronize the emulation with the external clock of the device.

With minor modifications, the HCSE can also be run in batch mode using the VSM SUBMIT command. In this way, the user may run Monte Carlo tests to estimate error statistics such as mean times between failures.

Additional capabilities are described in the HCSE Applications Guide.

1.0 APPENDICES

1.1 Specific Hardware And Software Requirements

Computer: Digital Equipment - VAX 11-780

Storage: Minimum one and preferably three disk drives

Terminals: DEC-supported terminals or fully compatible equivalent (VT-52) or (VT-100). FSM source files may be developed on non DEC-supported terminals.

Hard-copy Devices: No specific requirements. Fortran unit 6 is used by SUMMARY for output. SYS$OUTPUT is used by other display programs; this may be redirected to a hard-copy device at the user's discretion (see Section 4.4).

Software: VMS Version 2.7
VMS utilities and libraries are required.

The following files are required:

Command Files (defined under system logical name HCSE_LIBRARY)

FOREIGN.COM BEGIN.COM SMERGE.COM

Libraries (defined under system logical names PRX$SYNOPSIS_0 through PRX$SYNOPSIS_4)

PRAXIS_LIBRARY BP_LIBRARY (references FORTRAN library) CM_LIBRARY, which includes

VAXDEF.OBJ
VAXDEF.SPS
SHRMEM.OBJ
SHAREOUT.OBJ
SHAREOUT.SPS
SHAREDMEM.SPS
VAXRUNTIM.SPS

Executable images (under system logical name i 21;HCSE_LIBRARY)

PARSER.EXE DICTION.EXE BUILDER.EXE DISPLAY.EXE FORCEX.EXE SIMLIST.EXE SUMMARY.EXE

1.2 Creating State Machine Modules From State Machine Descriptions

Using the example of Section 5, this appendix illustrates procedures for converting various state machine descriptions into the FSM file format. The most common means for describing finite state machines with a small number of states is the state transition diagram. The machine characterizing the emulation of Section 5 is shown in Figure 10.

The contents of the balloons are the value of the state. On the arrows are the values of the input(s), followed by the values (or changes in) the output values. The name of the input, output, and state variables are usually implicit in this representation. The integer variable "count" is not finite-valued and the standard notation does not apply, in a strict sense, to this example. However, the values of "count" are intended to be finite in this example. With this foreknowledge, "count" could then be regarded either as a second state variable or as both an input and output variable. The latter point of view is taken in this example. Note that in machine COUNT2 there is an unconditional state transition from <NUL> to RUNNING without an output change.

A state-machine can always be represented in terms of a next state and output function, defined on the set of all possible values of current state and input. For the present example, these are specified by Table 2.

Since the current value of "count" does not influence the next

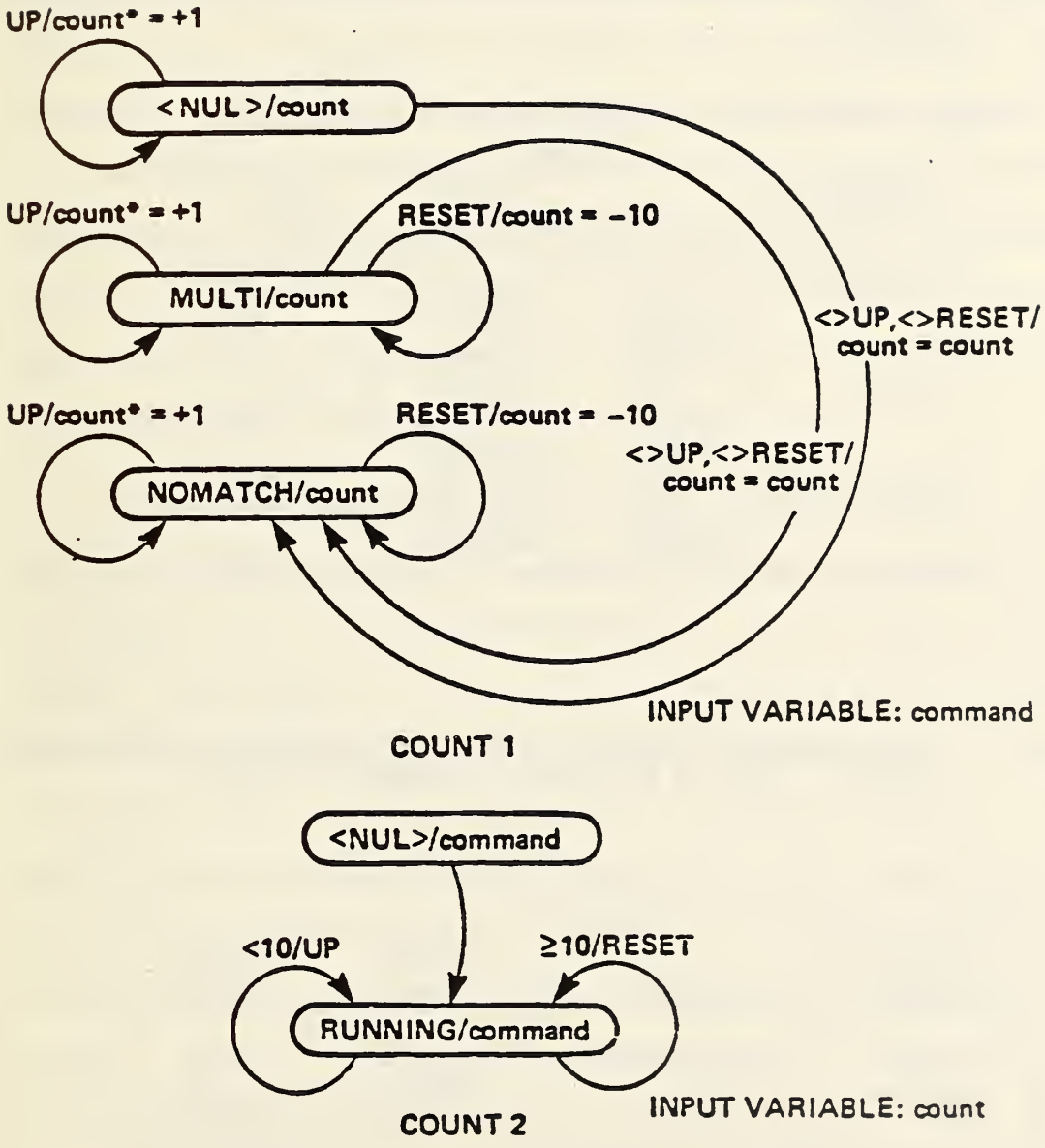


FIG. 10. STATE MACHINE DIAGRAM FOR EXAMPLE

TABLE 2. (A) ENUMERATED NEXT STATE AND OUTPUT FUNCTIONS FOR COUNT1

COUNT 1			
CURRENT STATE	INPUT	NEXT STATE	NEXT OUPUT
<NUL>	UP	<NUL>	[ ]+1
	RESET	<NUL>	-10
	OTHER	NOMATCH	[ ]
MULTI	UP	MULTI	[ ]+1
	RESET	MULTI	-10
	OTHER	NOMATCH	[ ]
NOMATCH	UP	NOMATCH	[ ]+1
	RESET	NOMATCH	-10
	OTHER	NOMATCH	[ ]

TABLE 2. (B) ENUMERATED NEXT STATE AND OUTPUT FUNCTIONS FOR COUNT2 NOTE: [ ] DENOTES CURRENT OUTPUT)

COUNT 2			
CURRENT STATE	INPUT	NEXT STATE	NEXT OUTPUT
<NUL>	any integer	RUNNING	[ ]
RUNNING	< 10	RUNNING	UP
	≥ 10	RUNNING	RESET

state of COUNT1, it has not been shown in a separate column; its value is used implicitly in the last column. Note that neither "count" nor "command" variables are initialized, making this formally a non-deterministic machine. The state table is merely a definition of the next state and output functions by denumeration. Sometimes, current state/input conditions which leave the current state and next output unchanged are eliminated from the table; the last row of the table for COUNT1 is the only case where this occurs.

The relationship of the relay ladder diagram and state machine representation is somewhat difficult to describe. In the case where all quantities are binary, each "rung" of a relay ladder implements a binary-value function of the current state and input. In the case where states, inputs and/or outputs are string-valued, it is necessary to code the values of the state, input, and output as binary numbers and then represent the next-state and output functions bit-by-bit. Other elements of relay ladder networks include counters (typically with toggle inputs), preset timers, and switches. The essence (but not the details) of the preceding example can be represented by the diagram of Figure 11.

At the left is a bus with binary 1 (or "true"). The coil (open circle) at the right provides the value of the variable denoted

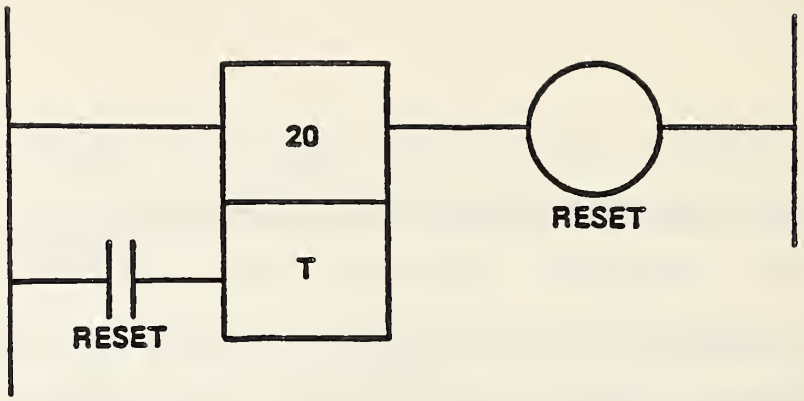


FIG. 11. RELAY LADDER DIAGRAM FOR EXAMPLE

"RESET" (this would typically be assigned a coil number in practice). The partitioned rectangle represents a preset timer with preset value 20 [=10-(-10)] from the example). "T" represents the register in which the elapsed time since last reset is stored; this is comparable to (count+10) in the example because by convention, the preset timer resets to zero. The upper left line is the control line; the clock counts whenever it is "true". The lower line is the reset line, which resets the timer to zero whenever its value is true. The intended effect of this rung is that the preset timer counts to 20 and then resets itself, since the normally open relay on the reset line represents the same coil designated "RESET" on the right. This formal configuration may not be legal in the notation used by some manufacturers, but it illustrates the nature and intent of the example. Module COUNT1 is represented, conceptually, by the timer while module COUNT2 is represented by the coil.

The FSM modules COUNT1 and COUNT2 corresponding to the example are shown in Figure 5. By examination of Table 2(a), the next state is seen to be actually independent of the current state, so that the "conditions" lines do not involve the current state in COUNT1. In module COUNT2, there is one "conditions" line for each line of the state table. The "actions" line computes the next state and output. The "multimatch" and "nomatch" lines are action lines for defining the next state and output under conditions that violate the normal rules of state machine representation, viz., that all input combinations be considered in the state table (so that the next state and output maps are functions rather than partial functions), and that the values of the next state and output functions (by definition) be singly-defined.

1.3 Praxis Primer

UCRL-52957 Rev. 1

An introduction to Praxis

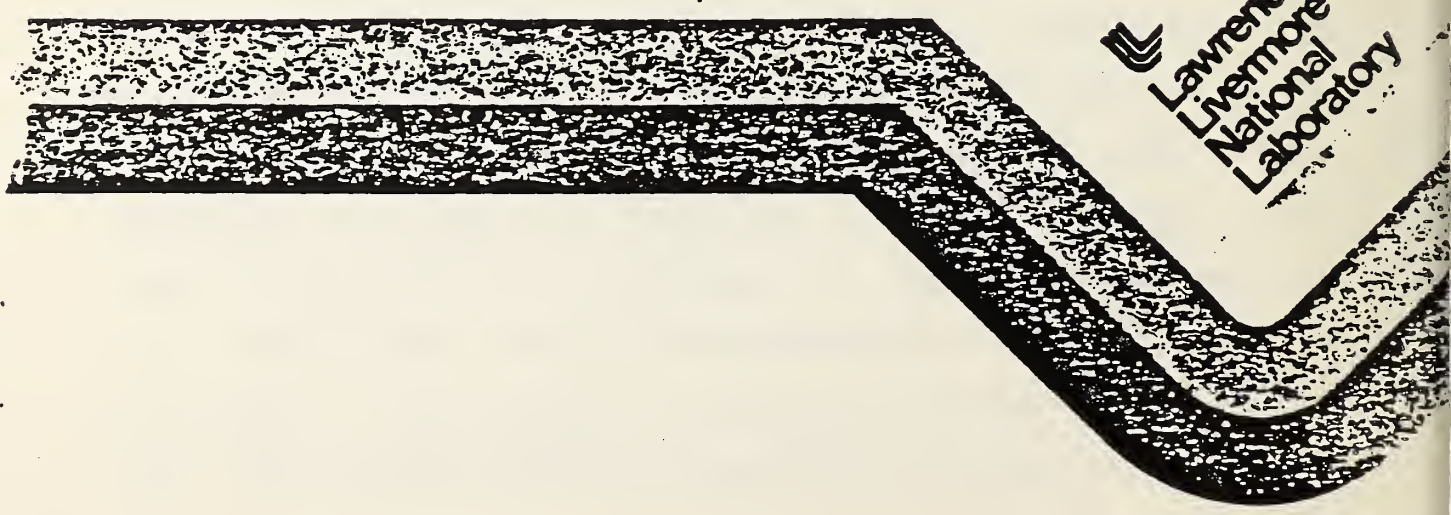
James R. Greenwood

Arthur Evans, Jr.

C. Robert Morgan

Michael C. Zarnstorff

December 1980



An introduction to Praxis

ABSTRACT

Praxis is the practice of the programming art, science, and skill. It is a high-order language designed for the efficient programming of control and systems applications. It is a comprehensive, strongly typed, block-structured language in the tradition of Pascal, with much of the power of the Mesa and Ada languages. It supports the development of systems composed of separately compiled modules, user-defined data types, exception handling, detailed control mechanisms, and encapsulated data and routines. Direct access to machine facilities, efficient bit manipulation, and interlocked critical regions are provided within Praxis.

Keywords: Praxis, high-level control language, compilers, real time.

Section 1

INTRODUCTION

This report describes the control-system implementation language Praxis, which has been developed in the Laser Fusion Program at the Lawrence Livermore National Laboratory (LLNL) for control applications. It serves as an introduction to the language so that the reader can get a *feel* for what the language is and find out if it is applicable to the reader's needs.

Most of the report consists of graduated examples that provide an overview of the language. The definition and details of the language can be found in the *Praxis: Language Reference Manual* and in other companion reports that follow the publication of this report.

Section 2

DEVELOPMENT HISTORY

In the summer of 1978, it became apparent in the laser fusion program at LLNL that we needed a control-oriented language for use in programming the control system of the Nova laser system. Our experience in developing the laser control system for Shiva, consisting of 55 processors, clearly indicated that if a controls-oriented programming language were available we could save considerable time and effort with respect to Nova.

After carefully evaluating potential languages, including DOD's current development of Ada, we chose to implement Praxis. Although Ada would meet our needs, it would not be available in time for Nova (compilers had to be available before the mid-1980's to meet the needs of the Nova controls programming). In retrospect, our selection of Praxis proved correct, since a Praxis compiler now exists and is in use while the more ambitious Ada development is still ongoing.

The development of Praxis originated from an initial study by Bolt, Beranek, and Newman (BBN), Inc., funded by the Defense Communications Agency (DCA), to determine the requirements of a language for communications programming. The result of that study (BBN Report 3261) concluded that no current language fulfilled the rigorous needs of communications programming.

The DCA then funded BBN to design an appropriate programming language. This resulted in a preliminary design of the COL language described in BBN Report 3534, May 1977 (A. Evans, C. R. Morgan). Also, the DCA funded BBN to design a compiler described in BBN Report 3533, May 1977.

In January 1979 LLNL funded BBN to augment the design of COL and to implement a COL compiler for the PDP-11 series of computers from Digital Equipment Corporation. With the clarification of the Nova controls design and schedule, BBN's work has been expanded to include the development of a VAX/VMS native-mode compiler, documentation, additional language design, and a high-level input/output package. BBN is scheduled to complete their work by fall 1980, with the delivery of documented operational compilers for Praxis, on both the PDP-11/RSX-11 and VAX/VMS systems, written in Praxis.

In January 1980 we changed the name of the language from COL to the current Praxis. We felt that the language had evolved significantly from that of the original COL study and that a new name would better reflect its power.

In March 1980 the preliminary PDP-11 compiler successfully passed two critical milestones. The first milestone was that the compiler, which is written in Praxis, had to compile itself successfully on the PDP-11/RSX-11M system. This would demonstrate that the compiler was self-supporting on the PDP-11 systems, and that the bulk of compiler was correctly implemented.

The second milestone was the implementation of a Nova controls application of the language, for a ROM-based LSI-11 processor. A 2000-line assembly-language, stepper-motor control program had to be recoded in Praxis, compiled, and *burnt* into read-only memory (ROM). This would demonstrate that the language was indeed powerful enough to replace detailed, assembly language sequences and that the compiler correctly implemented the controls-oriented features.

Section 3

INTENDED APPLICATIONS

Praxis is designed for programming control and communication applications. It is also useful for system programming applications, which require many of the same language facilities found in Praxis. All these applications impose stringent requirements on programming in such areas as

- Efficiency of object code.
- Direct access to machine facilities.
- Efficient bit manipulation.
- Complex data and control structures.
- Large programs developed by a team.
- Maintenance and upgrades.

The programming of these applications requires detailed control of the compiler-produced code, the optimization, the variable allocation, and the run-time support. In these applications, it is important for the programmers to explicitly control *exactly what is going on*.

Section 4

DESIGN GOALS

The design goals of Praxis are based on the requirement of the language being a useful tool for programming control applications. Consequently, the goals may be stated as follows:

- Efficiency: first of the compiled code, then of the compiler.
- Readability: particularly more important than writability.
- Completeness: in the sense that
  - it must be possible to program all of any one application in Praxis without recourse to assembly language.
  - it must be possible to write the compiler for Praxis in the Praxis language.
- Portability: Praxis should be reasonably machine-independent.
- Modularity: it must be possible to program large projects within Praxis, requiring separate compilation of modules and configuration control.
- Usability: primarily used by experienced programmers, so that the ease of learning Praxis is less important than the ease of using Praxis.

The primary requirements for control applications are efficiency of the compiled code, completeness, and portability. Praxis must produce programs that make effective use of hardware resources directly controlled by the programmer. Also, the programs should be as portable as possible between machines. In general, the language features are portable but, where machine-dependent parts are necessary, they are as conspicuous as possible. For example, the programmer can override the language's type-checking mechanism, but it is easy to see when this is being done.

The requirement for efficiency has had one other impact on the language design. All proposed features and facilities have to be scrutinized for the run-time and the compile-time efficiency of their implementation. No matter how desirable a particular feature might be, it had to be rejected if a reasonably efficient implementation could not be designed.

Section 5

LANGUAGE OVERVIEW

Praxis is a modern, block-structured, fully typed, algorithmic programming language in the tradition of Pascal. Its design has been influenced by the languages Simula, BCPL, Euclid, PL/I, Jovial, CS-4, Alphard, Mesa, and Bliss languages, as well as by the DOD's language development work and the proposed Ada language. In scope and power, Praxis most closely resembles Ada and Mesa.

Since the control environment differs in important ways from application to application and machine to machine, Praxis has features to handle these differences. High-level facilities that mask machine dependencies and foster machine independence (portability) usually prevent the use of exactly the programming capability needed for real-time, control applications programming. However, Praxis is a high-level language that has controlled access to machine dependencies.

Praxis is *strongly typed*. The programmer is given a collection of predefined types and has the ability to construct new types. Every variable, constant, parameter, and expression has a type. All types can be deduced at compile-time and the compiler requires that each value be used in a way that is consistent with the rules associated with its type. For instance,

it is a compile-time error to attempt to pass an integer parameter to a routine that requires a real parameter.

The language is *blocked structured*. Blocks are a method of packaging statements and declarations so that the scope of the statements is clearly specified and controlled. Praxis has more than 10 block-structured statements, each of which is delimited by an *XXX*/*endXXX* pair, where *XXX* represents the particular statement name. For instance:

```
for . . . . . endfor
if . . . . . endif
procedure . . . . . endprocedure
select . . . . . endselect
```

The block structuring also enforces a particular programming style that is more readable and maintainable than that of unstructured programming.

A simple example in the language is the matrix multiply of two N by N matrices named *SpecA* and *SpecB* and storing the result in *Spectrum*:

```
for I := 1 to N do
    for J := 1 to N do
        Spectrum [I,J] := 0
        for K := 1 to N do
            Spectrum [I,J] := Spectrum [I,J] + SpecA [I,K]* SpecB [K,J]
        endfor
    endfor
endfor
```

This example only makes sense within the scope of the declarations for the variables used. All the variables, except the one for loop indices, must be declared before use. Thus, the code above would be preceded by something of the form

```
declare
    N = 32                                  // constant
    SpecA : array [1..N,1..N] of integer    // an array variable
    SpecB : array [1..N,1..N] of integer    // an array variable
    Spectrum : array [1..N,1..N] of integer // an array variable
enddeclare
```

This declaration block could be written more concisely in various forms. One method would be to use a user-defined type for the array declarations, which then would ensure that the three arrays are all the same type and remain so with subsequent software maintenance. Thus, the declarations could take the form

```
declare
    N = 32                                  // a constant
    matrix is array [1..N,1..N] of integer  // a type
    SpecA : matrix                          // an array variable
    SpecB : matrix                          // an array variable
    Spectrum : matrix                       // an array variable
enddeclare
```

Note that we have used the language's comment convention "//." which designates that all text to the right on the line is treated as a comment. Here, all language-reserved words are boldface in the examples, but no distinction is made in actual programs.

Another example is a simple exchange sort in which a values array is sorted into ascending order:

```
declare
    N = 100                                  // a constant integer
    data : array [1..N] of integer           // an integer array variable
    done : boolean                           // a true/false variable
enddeclare
... code to store values in data ...
repeat
    done := true                             // nothing out of order found
    for K := 2 to N do
        if data [K-1] > data [K] do
            swap (data [K-1], data [K])      // if out of order, exchange them
            done := false                    // not done yet
        endif
    endfor
until done
```

The **repeat** block-structured statement is the exception to the ending syntax rule, in that the **until** is the end for the **repeat** block. The **repeat/until** has the semantics that the included statements are executed repeatedly until the expression after the **until** is true. Other looping constructs are available in Praxis, including the **while/endwhile**, and four forms of **for/endfor**.

A more detailed control programming application is shown below. It directly reads a hardware input/output device on a PDP-11 computer in a multi-process environment. In this example, the resource (i.e., I/O device) is protected by the interlock variable *padlock* in a critical region. Another process with similar code, using the same resource, cannot preempt the critical-region code sequence.

```
Declare
    status : location (8!176420) volatile logical    // status register
    datum  : location (8!176422) volatile char       // input register
    padlock : static interlock                       // exclusion variable
    temporary : char
enddeclare
...
Region padlock do                                    // wait for device ready
    Repeat until (status and 8#200) <> 8#0           // read the character
    temporary := datum                               // lock the interlock
endregion                                            // unlock the interlock
```

The attribute **volatile** on the variables *status* and *datum* informs the compiler that the variables must be referenced directly each time they are mentioned in the program, and no optimizations are to be performed on these variables. It allows variables to be used as I/O registers, as above, as well as to be used in shared memory.

The **location** attribute informs the compiler to place the variable in the physical address specified by the octal (8!) integer constant in the parentheses. The variable is static and always resides at that location. The **static interlock** is at a fixed location determined by the compiler.

The **logical** predefined data type may be thought of as a bit-string data type on which bit-by-bit operations may be performed. In the **until** clause, a bit in the *status* variable is tested by the bit-by-bit **and** with the octal (8#) logical constant and comparison to a logical zero.

A more complex application, which demonstrates the ability in Praxis to bypass the strong typing (when desired), is the sequence that extracts the exponent value from a real number on the PDP-11:

```
Declare
    scale : real                                    // floating point variable
    power : integer                                 // signed integer variable
    temporary : logical                             // 16-bit bit-string variable
enddeclare
... code assigning value to scale ...
    temporary := ((force logical (scale)) rsh 8) and 8#177
    power := integer (temporary) - 8!100            // make -N to N
```

The force explicitly overrides the type-checking mechanism and specifies that the variable *scale* is to be handled as a logical in this expression. The logical value (i.e., 16 bits) is shifted right 8 bits and masked with the logical constant. *Temporary* is assigned the resulting value that was the exponent of the real variable *scale*. The logical value is then converted to an integer and stored in the variable *power*.

Note the distinction between force and type conversion; force informs the compiler to treat a variable as a particular type regardless of its actual type; conversion causes the variable to be converted to the desired type.

Another application of type conversion is shown in the function *upper*, which converts a possible lower-case letter to an upper-case letter:

```
function Upper (inchar:char) returns outchar:char
    if inchar < $a or inchar > $z do
        outchar := inchar                           // set returned value
        return                                      // exit if not lower-case letter
    endif
    outchar := char (integer (inchar) - 8!40)       // convert to upper-case
endfunction |Upper|
```

The previous function example utilized the return statement for explicit exit from a routine (i.e., procedure or function). This statement is one of several such statements that eliminates the need for a GOTO in the language. An important *feature* in the language is the *lack* of the GOTO statement. The following example uses two other control flow statements, together with block labeling, to program an application that normally requires a GOTO statement.

```
primary : For index := 0 to Bound do                // labeled statement
            size := Motor_size index                // assignment
            While Motor [index] = on do             // inner loop
                if size < mid_size do               // conditional statement
                    loop primary                    // iterate for loop
                orif size < max_size do             // alternative
                    break primary                   // exit for loop
                otherwise                           // default alternative
                    Slew_motor (index)              // procedure invocation
                endif                               // end of alternatives
            endwhile                                // end of inner loop
          endfor |primary|                          // end of labeled block
```

The loop statement above causes the for loop iteration to occur: that is, it acts like a GOTO the for, which causes the iteration count of the loop to be incremented, the test for completion to be performed, and the for block to be executed if the iterations have not been completed. The break statement on the other hand is a block exit statement. In the above case, it exits three levels of blocks: the if, while, and for, and execution continues after the endfor. Labels can only appear on blocks (at the beginning and end) and are only used with the break, loop, and retry (in critical regions) statements.

The statement sequence above would have had to be preceded by a declaration in which the variables, types, and constants are declared. All items must be declared before their use. The declaration for the above could be

```
Declare
    min_size = 0                          // a constant
    mid_size = 25
    max_size = 50
    size : integer initially min_size     // a variable
    bound : integer initially 0
    Onoff is [on, off] initially off      // an enumeration type
    Motor_size : array [0..9] of integer  // array variable
    Motor : array [0..9] of Onoff         // array variable
enddeclare
```

Notice the use of initialization clauses on variable and type declarations, which allow for variables to be declared with initial values. For instance, the variable *bound* is declared with the initial value zero, and the variable *motor* is declared as an array of enumerated data values, initially all elements being the value *off*. The declaration form is declares new data types and is discussed more fully below.

The break and loop example above also introduced the multiarm if statement that allows the programming of a *branch-tree*. Only one arm of the statement is elaborated on each iteration of the while loop, depending on the boolean expressions in each arm. Any number of orif clauses may be present, and the otherwise clause is optional. Thus, the forms below are valid if statements:

```
if (x = 0) or (y = 15) do
    ....
endif
if x = y do
    ....
otherwise
    ....
endif
```

Another form of flow control statement in Praxis is the select statement, which selects a sequence to elaborate from a set of cases according to a selection expression. For example:

```
Declare
    subsystem is [power, align, beam, target]   // a type
    system : subsystem initially beam           // a variable
enddeclare
```

```
....
select system from
    case power : Print ("Power subsystem")
    case align  : Print ("Alignment")
    default     : Print ("Others")
endselect
```

Only one of the *Print* procedure invocations is executed, depending on the value of the enumerated variable *system*. Note that the default clause will be executed for any values other than *power* or *align*. The strong typing and declarations ensure that the only other enumerated values the system can take on are *beam* and *target*.

Another control application that can be run on the PDP-11 uses data structures, procedure variables, and interrupt procedures to quickly and easily program an application that normally must be done in assembly language:

```
interrupt procedure clock_service ( )
        ticks := ticks + 1
endprocedure {clock_service}

declare
    vector is structure
        routine : interrupt procedure ( ) initially clock_service
        status : logical initially 8#340
    endstructure
    clock : location (8!100) vector
    ticks : static integer initially 0
enddeclare
```

The variable *ticks* gets incremented for each interrupt from the line clock on the PDP-11.

Note that because the interrupt procedure is executed asynchronously, communication with the other code must be done through static variables. Only one copy exists of any static variable.

The user-defined structure data-type *vector* has two fields: the first is the *routine*, which is of type Interrupt Procedure and is initialized to be the *address* of the clock service routine; the second field is a logical (i.e., bit-string) variable, which is set to the value desired for the processor status word. The actual declaration and positioning of the clock vector are accomplished by the variable declaration *clock* and the location attribute.

The above sequence would most likely be used in conjunction with a read routine of the form

```
function Read_ticks ( ) returns t : integer
    t := ticks
endfunction {Read_ticks}
```

The empty parentheses (i.e., ( )) denote a routine with no parameters and would be invoked with the form

```
count := Read_ticks ( )                    // get # of ticks
```

The interrupt-procedure example utilized the structure data type (i.e., the user-defined *vector*) and the procedure data type. These data types are two of the predefined data types in the language, all of which are listed below:

```
Discrete types
    integer         - signed
    cardinal        - unsigned integer
    char            - ASCII character
    boolean         - true/false
    enumeration     - programmer-specified values
Control types
    interlock       - locked/unlocked
    logical         - bit string
    pointer         - pointer to a typed object
Floating types
    real            - floating point
    long real       - double precision real
Aggregate types
    array           - array of any type, access by index
    structure       - various type components, access by name
    set             - set of discrete type
Routine types
    procedure       - typed procedure variables
    function        - typed function variables
Other types
    general         - union of all types (used as formal parameter)
    descriptor      - type descriptor
```

User-defined data types may be characterized in terms of the predefined types or other user-defined types. The is form declares a user-defined data type. The semaphore in the example below is a user-defined data type. *Sync* is a variable of type semaphore:

```
declare
    semaphore is structure                  // type decl
        lock : interlock
        count : integer initially 0
    endstructure
    Sync : semaphore                        // a semaphore variable
enddeclare
```

This method for synchronization was proposed by Dijkstra in 1968. The semaphore is a special variable that can be manipulated only by the primitives Wait (also called the P operator) and Signal (also called the V operator), defined as follows:

```
procedure Wait (Sem : inout ref semaphore)      // P operator
    Region Sem.lock do                          // protect count access
        if Sem.count  = 0 do                    // check count value
            retry                               // loop, unlock, a relock lock.
        endif                                   //
        Sem.count -= 1                          // decrement count
    endregion                                   // end of critical region
endprocedure |Wait|                             // return from procedure

procedure Signal (Sem : inout ref semaphore)    // V operator
    Region Sem.lock do                          // enter critical region
```

```
        Sem.count °= +1                    // increment count
    endregion                              // exit critical region
endprocedure |Signal|                      // exit from Signal
```

The *Wait* procedure allows a process to delay while waiting for an event to occur. The *Signal* procedure is used to signal another process that an event has occurred. In the above example, it is assumed that the semaphore would be shared between two processes, and each process would have its own copy of the *Wait* and *Signal* procedures. The interlock is utilized to guarantee *atomic* access to the semaphore count without worrying about actual code sequences.

The form "°=" assignment statement can be read as *transformed by*. Thus, the statement

```
Sem.count °= +1
```

increments the *count* field of the semaphore passed as an argument to *Signal* and is equivalent to the statement

```
Sem.count := Sem.count +1
```

The formal parameter specification on *Wait* (and *Signal*) explicitly specifies that the actual parameter be passed by **Ref** (i.e., reference) and that the parameter will be both read (i.e., **in**) and written (i.e., **out**). Parameters may be passed by **Ref** or **Val** (i.e., value, by copy) with the default being by **Val**. The programmer would usually specify by **Ref**, for large aggregates, in the interest of efficiency. The data-passing direction can be specified as **in**, **inout**, or **out** with the default being **in**. The compiler checks at compile-time to ensure that the usage of the parameter, within the routine, is consistent with the passing direction.

The *semaphore*, *Wait*, and *Signal* definitions can be encapsulated within a **Module** for separate compilation, or for data abstraction, or for both. Thus, the definition module would be

```
Module Semaphore_package
Export semaphore, Wait, Signal
Declare
    semaphore is hidden structure
        lock  : interlock
        count : integer initially 0
    endstructure
enddeclare
Procedure Wait (Sem : inout ref semaphore)
    . . . . .
endprocedure |Wait|
Procedure Signal (Sem : inout ref semaphore)
    . . . . .
endprocedure |Signal|
endmodule |Semaphore_package|
```

The declarations of *semaphore*, *Wait*, and *Signal* are made available by the **Export** to other modules (i.e., if this module was within another) or to other separately compiled modules that **Import** the declarations. Note that types, as well as data and routines, can be imported and exported.

The **hidden** attribute specified on this new declaration of the semaphore type implements what is referred to as an abstract data type. That is, the type name is known outside of the module, but the internal structure is unknown. Thus, an application program can import the type and declare and use variables of type semaphore without knowing the details of the structure. For instance:

```
Main Module Joe_Schmoe
Import semaphore, Wait, Signal from Semaphore_package
Declare
    Async : segment (control_area) volatile semaphore
    Bsync : segment (control_area) volatile semaphore
enddeclare
While true do                                   // infinite loop
    Wait (Async)                                // process synchronization
        ....
        ....
    Signal (Bsync)                              // process synchronization
endwhile
endmodule {Joe_Schmoe}
```

The **main module** allows the use of top-level code (i.e., code not within a routine) and is the main program or process, depending on the operating system employed. In the example, two variables, *Async* and *Bsync*, are declared, using the imported semaphore definition. These variables are then used with the *Wait* and *Signal* procedure calls to synchronize this process with other processes. Note that the language makes no assumptions about the run-time system; no tasking or multiprocess operations are built into Praxis. These facilities can be programmed in the language, or provided by existing operating environments.

The **segment** storage class on the declarations of *Async* and *Bsync* specify that the semaphores are static in a named (i.e., control_ area) data area. This data area can be associated with program sections or location counters (depending on the implementation) by means of the **%Segment** compiler directive. For instance, for a PDP-11/RSX-11M implementation, the directive

```
%Segment control_area = RW, D
```

creates a program section (i.e., PSECT) which can be controlled and positioned at link-time. **Segment** can be viewed as a named location.

The *Print* routine used in a previous example could be written as

```
Procedure Print (string : in ref array [1..?length] of char)
    For index := 1 to length do
        Put_character (string [index])
    endfor
endprocedure {Type}
```

The formal parameter specifies a flexible array of characters as the type of the parameter: this allows the arrays of characters of any length to be passed, with an implicit-size parameter *length*. A quoted string is considered an array of characters indexed 1 through N, where N is the number of characters between the quotes.

Flexible arrays can also be allocated from the free memory storage (i.e., heap) and accessed through pointers. The free memory is only utilized when the programmer explicitly

specifies it by the **allocate** and **free** operations. There is no implicit heap usage or *garbage collection* in the language, an essential requirement in real-time control applications. Data objects in the heap are referenced by pointers. For instance:

```
Declare
    node is pointer structure
        address : integer
        status : logical initially 8#201
        data : array [-3..2] of real
        next : node initially nil
    endstructure
    head : node initially nil
enddeclare
    head := allocate node (address : 8!177560)
    if head@.data [2] = 0 do
        ....
    endif
```

The *node* declaration is a pointer to a structure of the form shown. *Head* is a declaration of a pointer object, and the assignment statement creates an object within the heap and places the location of the object in the variable *head*. The field *address* will be initialized to the octal value 177560, and the field *status* will be initialized to the octal value 201 via the type initialization clause.

The object is referred to with the "@" operator. That is, since *head* is a pointer to a structure, then

```
head@                - whole structure
head@.address        - an integer field
head@.data[J]        - an element of a field
head@.next           - a field
head@.next@.address  - a field of an object pointed to by a field
```

are valid references. Note that the last reference only makes sense if the value in the *next* field points to something (i.e., not **nil**).

The node pointer structure allows a linked list to be allocated at runtime from the heap. The iterator form of the **for** loop is useful for stepping through such a list.

```
For p := head then p@.next while p <>nil do
    if p@.status and 8#200 <> 8#0 do
        ....
    endif
    ....
endfor
```

The pointer variable *p* is declared and is assigned the value from *head*: if the value is not **nil** then the body of the **for** block is elaborated. The expression between the **then** and **while** is the iteration expression that specifies the subsequent values of *p*.

Objects allocated from the heap must be explicitly returned with the *free* procedure, which has the form

```
Free (p, head)                          // release P and head
```

Free may be called with any type of pointer and any number of parameters.

An important consideration in real-time systems is the ability to handle abnormal conditions and catastrophic failures. In Praxis, this is accomplished with named exceptions and guard blocks. Both predefined and user-defined exceptions are available and can be caught with a **guard** block. Thus,

```
Guard
    .....
    X := Y/Z
    .....
catch
    case divide_zero : Print ("Whoops")
endguard
```

would catch any divide-by-zero exception in the code between the **guard** and **catch** phases, or in any nested routines invoked from within the code. When and if a named exception occurs, the first (deepest) dynamically nested catch case for the named exception is elaborated. The **catch** clause can specify various named exceptions as well as use a **default** clause (i.e., all others).

Guard blocks may be used to contain exceptions in a large program or to catch an exception from a localized section. For instance, the Praxis input/output package uses exceptions for abnormal condition handling:

```
Import Open, Open_error, file, Name_error from IO_package
Declare
    myfile : file
enddeclare
Guard
    Open
        with
            name   : "DB3: [Shiva] Test.dat"
            file_id : myfile
        ___access : default_access
        endwith
catch
    case open_error : Print ("Bad I/O")
                      raise Bad_IO
    case name_error : Print ("Bad filename")
endguard
```

The *Open* procedure invocation is surrounded by a **guard** block; the procedure upon detecting an error will **raise** the exception *open_ err* that is declared in the *IO_ package* Control is transferred to the **case** clause in the **catch** block for the exception named The clause is then elaborated. In the *open_ err* case, a routine is invoked and then a user-defined exception is explicitly raised, and elaboration continues in a higher-level guard block. For the *name_ err* exception, the case clause is elaborated and elaboration continues after the **endguard**. If no exceptions are raised within the *open*, then elaboration continues after the **endguard**.

The *Open* example also introduced an alternate procedure invocation, using named formal parameters and the list (i.e., with-endwith) format. The named parameters allow the use of optional parameters and parameter specification in any order. The name on the left of the colon (:) is the name of the formal parameter, and the name on the right is the actual parameter of the invocation. The declaration of the *Open* procedure could be of the form

```
procedure Open
    param
        file_id : in val file
        name : in ref array [1..?N] of char
        access : in val set of access_type
        window : optional in val 8 bit integer
                    initially 0
        share : optional in val set of sharing
                    initially empty_share
        . . more optional parameters . .
    endparam
. . procedure body . .
endprocedure  Open
```

Only formal parameters declared as optional may be omitted in any actual invocation. Each formal parameter specified as optional must have a default value specified by the initially clause.

The Guard example introduced two procedures from the Input/Output package. The package is implemented as a series of procedures, functions, and abstract data types written in the language. Each implementation will have slightly different I/O packages, tailored to the particular operating environment. Under the PDP-11/RSX-11M and VAX-/VMS operating systems, the I/O package supports the full RMS-11 capabilities including indexed files. The standard, I/O-related, encapsulated data types are

file	record	stream	attribute

and some of the routines are

Create	Open	Close	Extend
Display	Erase	Connect	Disconnect
Find	Delete	Flush	Release
Get	Put	Rewind	Update

In addition, a set of conversion routines for the predefined data types are supplied, which convert to/from ASCII text.

Other packages are supplied with implementations, or are supplied as interfaces to existing packages in other languages. Praxis routines can invoke other language subroutines and functions, or they may be called from other languages. For instance, a Fortran mathematics package would be defined as

```
Module Math_package
Export Sin, Cos, Log
Fortran Function Sin (X : real) returns Y : real
. . . .
endfunction |Sin|
. . . .
endmodule |Math_ package|
```

Other than the Fortran linkage, Praxis provides the linkages

```
Inline    - Place routine code in place of invocation
Interrupt - PDP-11 Interrupt service routine
```

Different compiler implementations could supply additional linkages.

An important feature that is necessary in the control environment is the ability to control the actual code generated for differing applications: for instance, the ability to generate code that would reside in ROM. This control is supplied by means of both predefined and user-defined compiler variables (comp_var), in conjunction with compiler directives. For instance:

```
%define Author, three_D
%Set Author = "J R Greenberg"          // string comp_var
%Set Object_ROM                        // predefined comp_var
%Set three_D = true                    // user defined

Declare
    span is 0..5
    %if three_D or All_three
        data : array [span, span, span] of real
    %otherwise
        data : array [span, span] of real
    %endif
enddeclare
```

Compiler variables can be either boolean or string types and are explicitly declared and assigned to by the %Set compiler directive. The comp_var *Object_ROM* specifies that the code generation should be such that the code and constant data can be *burned* into ROM. The %if-%otherwise-%endif allows conditional compilation under control of a boolean comp_var expression. The referenced comp_var values can be set either within program text or upon compiler invocation.

Another feature that needs mentioning is the ability to generate specific instructions or nonstandard calling sequences. This is provided by the block-structured code statement shown below for a PDP-11 application:

```
Procedure Trigger (X : integer)
    Declare
        timer : static integer
        index : integer
    enddeclare
    code "PDP-11" do
        MOV #33, index(SP)          // set a count
        MOV X(SP), R1               // pass parameter
    LP: INC timer                   // strobe
        TRAP                        // go to another routine
        DEC index(SP)               // count
        BNE  LP                     //
    endcode
    if timer = X do
```

```
        endif
    endprocedure {Trigger}
```

The concluding example outlines a simple task processor, using arrays of procedure variables and the set data type:

```
Declare
        number is integer range 0..5                    // range of integers
        active is set of number                         // set of integers
        active_tasks : static active initially active ()  // a set variable
        task is procedure ()                            // procedure type
        task_list : static array [number] of task       // list of possible tasks
enddeclare
....
Procedure Activate (task_id:number)
        Active_tasks *= + Active (task_id)              // place in set
endprocedure {Activate}
....
For index in active_tasks do                            // scan all active tasks
        task_list [index] ()                            // invoke task
endfor                                                  //
```

The set data type in the declaration of *active* is used as an attribute associated with each *task*. The set has six possible members denumerated by the values 0 through 5. Sets can be of any discrete type and can be arbitrarily large (i.e., limited by memory size of machine). The *active ()* after the initially clause and in the assignment statement is the set constant constructor, which allows items from the set to be included or removed. The For statement iterates through the set of *active_tasks* and will invoke any *active* task.

Section 6

SUMMARY

The preceding section, although introducing many features of the Praxis language, is by no means exhaustive. Some features have not been mentioned, and others have only been partially described. The full language is described in *Praxis: Language Reference Manual* and the *Programming in Praxis* manual.

The Praxis language is specifically within the state of the art of language design, particularly designed for control and system implementation needs. Complex language features, such as generic procedures, overloading of operators, and parallel processes, have been intentionally left out. We felt that these concepts were either not understood enough to be incorporated at this time, or that they need not be part of the language.

In conclusion, Praxis is an extremely powerful, modern programming language that goes beyond Pascal and is available today.

ACKNOWLEDGMENTS

The original language was designed by Arthur Evans, Jr., and C. Robert Morgan of BBN in 1977. Additional language design in 1979 by Evans and Morgan was augmented by James R. Greenwood (LLNL) and Michael C. Zarnstorff (University of Wisconsin). The final language design in 1980 was developed by the above individuals, with contributions from Earl Killian (BBN), Graeme Williams (BBN), and W. Nowicki (Stanford University).

The continued support of the management of the laser fusion program and the Nova laser project at LLNL, in particular J. L. Emmett, J. F. Holzrichter, R. O. Godwin, and W. W. Simmons, is gratefully acknowledged. The encouragement and support of H. Ahlstrom and L. Coleman of the fusion experiments program at LLNL is also greatly appreciated.

The tremendous effort by F. Holloway in developing the first application program in Praxis for the Nova control system is hereby acknowledged. His patience with early compiler releases, his persistence in developing the application acceptence test, and his constant enthusiasm were invaluable to the success of the project.

Additional thanks go to G. J. Suski, P. Rupert, and the controls development group at LLNL for their willingness to attempt the project and suffer through the preliminary versions of the product.

Also, the dedicated support and documentation efforts by W. Nowicki was essential. In particular, his work on the *Programming in Praxis* manual came at a critical time.

The documentation and support role of J. Walker and R. Shapiro at BBN was extremely valuable. J. Walker created the *Language Reference Manual* in a short period of time from an everchanging definition.

BIBLIOGRAPHY

Many languages are identified in the body of this report without specific references. Citations are as follows:

Ada	(Ichbiah-79A-79B)
ALGOL-60	(Naur-63)
ALPHARD	(Wulf-76)
BCPL	(Richard-69), (BBN-74)
BLISS	(Wulf-71)
CS-4	(Intermetrics-75)
EUCLID	(Lampson-77)
FORTRAN	(FORTRAN-76)
IMP	(Irons-70)
JOVIAL	(Shaw-63)
Mesa	(Mitchell-79)
Pascal	(Jensen-74)
PL/I	(IBM)
Simula	(Dahl-70)

(BBN-74)

BCPL Manual, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts (1954).

(Brinch-Hansen-72)

P. Brinch Hansen, "Structured Multiprogramming," *Comm. ACM* 15, 7, 574-578 (1972).

(Brinch-Hansen-73)

P. Brinch-Hansen, *Operating Systems Principles*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1973).

(Dahl-70)

O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Common Base Language*, Norwegian Computing Center, Publication S-22 (1970).

(DoD-77)

"Department of Defense Requirements for High-Order Computer Programming Language—Ironman," January 14, 1977.

(Evans-76)

A. Evans, Jr., and C. R. Morgan, *Development of a Communications Oriented Language*, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 3261 (1976)

(Evans-77)

A. Evans, Jr., and C. R. Morgan, *A Communications Oriented Language (COL): Language Design*, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 3534 (1977).

(Evans-79)
A. Evans, Jr., C. R. Morgan, E. S. Roberts, and E. M. Clarke. *The Impact of Multiprocessor Technology on High-Level Language Design*. Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 4188 (1979).

(Fisher-76)
D. A. Fisher, "A Common Programming Language for the Department of Defense—Background and Technical Requirements," Institute for Defense Analysis, Paper P-1191, June 1976

(FORTRAN-76)
"Draft proposed ANS FORTRAN," *ACM Sigplan Notices* 11, 3 (1976) (entire issue).

(IBM)
"PL/I Language Specification," IBM Corporation, ANSII Standard for PL/I, Subset G, Form GY33-6003-2 (undated).

(Ichbiah-79A)
J. D. Ichbiah, J. Heiard, O. Roubine, J. Barnes, B. Krieg-Brueckner, and B. A. Wichmann, "Rationale for the Design of the Ada Programming Language," *ACM Sigplan Notices* 14, 6 (1979).

(Ichbiah-79B)
J. D. Ichbiah, J. Heiard, O. Roubine, J. Barnes, B. Krieg-Brueckner, and B. A. Wichmann, "The Preliminary Ada Language Reference Manual," *ACM Sigplan Notices* 14, 6 (1979).

(Intermetrics-75)
CS-4 Language Reference Manual and Operating System Interface, Intermetrics, Inc., Cambridge, Massachusetts, Report IR-130-2 (1975).

(Irons-70)
E. T. Irons, "Experience with an Extensible Language," *Comm. ACM* 13, 1 (1970).

(Jensen-74)
K. Jensen and N. Wirth, *PASCAL User Manual and Report* (Second Edition), Springer-Verlag, Berlin (1974).

(Knuth-73)
D. E. Knuth, *A Review of Structured Programming*, Stanford University, Stanford, California, Computer Science Department, Report STAN-CS-73-371 (1973).

(Knuth-74)
D. E. Knuth, "Structured Programming with Goto Statements," *Computing Surveys* (December 1974).

(Lampson-77)
B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchel, and G. J. Popek, "Report on the Programming Language EUCLID," *ACM Sigplan Notices* 12, 2 (1977) (entire issue).

(Mitchell-79)
J. G. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual V5*, Xerox Corporation, Palo Alto, California, Report CSL-79-3 (1979).

(Morgan-77)
C. R. Morgan and A. Evans, Jr., *Communications Oriented Language (COL): Language Implementation*, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, Report No. 3533 (1977).

(Naur-63)
"Revised Report on the Algorithmic Language ALGOL 60" (P. Naur, Ed.), *Comm. ACM* 6, 1, 1-17 (1963).

(Richards-69)
M. Richards, "BCPL—A Tool for Compiler Writing and Systems Programming," from *Spring Joint Computer Conference* (1969), pp. 557-566.

(Shaw-63)
C. J. Shaw, "A Specification of JOVIAL," *Comm. ACM* 6, 12, 721-736 (1963).

(Wirth-76)
N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1976).

(Wulf-71)
W. A. Wulf, D. B. Russell, and A. N. Haberman, "BLISS: A Language for System Programming," *Comm. ACM* 14, 12, 780-790 (1971).

(Wulf-76)
W. A. Wulf, R. L. London, and M. Shaw, *Abstraction and Verification in ALPHARD: Introduction to Language and Methodology*, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Department of Computer Science (June 1976).

(Zahn-74)
C. T. Zahn, "A Control Structure for Natural Top Down Structured Programming," from *Symposium on Programming Languages*, Paris, France (1974).

BJ/jvb

U. S Government Printing Office. 1981/1-789-002/5531

Appendix

LANGUAGE SYNTAX

Backus-Naur Form (BNF)

Here, we describe the context-free syntax of the language, using a variant of the Backus-Naur Form (BNF). In particular, we adhere to the following conventions in the BNF representation:

- Lower-case words, perhaps containing underscores, denote syntactic categories, such as:

  function_list
  relation_operator
  linkage

- Boldface words denote reserved words, for example:

  select
  function
  or

- Square brackets enclose optional items. A quoted square bracket means that it is part of the syntax (i.e., array subscripts and enumerations).

  endif [[label]]
  [mode] function ....
  [access_mode] **structure** ....

  array '[' subscript,...']' of type
  for ID **in** '[' enumeration,...']'

- Repeated items are represented by a delimiter followed by three dots. Thus, a list of identifiers could be designated by

  identifier,...

  where the comma is the repeat delimiter. Thus, the BNF form,

  identifier_list := identifier,...

  means that the identifier list can contain one or more identifiers separated by commas. Another example is

  statement_list := statement;...

  where the semicolon is the delimiter.

- The syntax rules describing structured constructs in the language are presented in a form that is visually similar to their usage in programs. For example, the select statement is specified in the BNF as

```
select_statement := [label:]   select expression from
                                 [case case_literal,... : sentence;... ] ...
                                 [default : sentence;...]
                               endselect [label]
```

- Various syntactic items can be represented by the item prefixed by a qualifier corresponding to a category name. The prefix is intended to convey extra semantic information. For instance:

  module_identifier module_ID function_identifier

  are all equivalent to:

  identifier

- Some abbreviations used in the syntax description are

ID	identifier
expr	expression
spec	specifier
c_constant	compile-time constant
l_constant	link-time constant

- The slash (/) is used to delimit various cases of a BNF production. It can be read as "or."

  Thus:

```
declaration ::= procedure_declaration
              / function_declaration
```

  is just shorthand for

```
declaration ::= procedure_declaration
declaration ::= function_declaration
```

Syntax Definition

```
module declaration   ::= [main] module module_ID segment_list
                           export ID,... [to module_ID....] ;...
                           sentence;...
                         endmodule [[module_ID]]

module_ID            ::= ID / module_ID.ID

sentence             ::= statement / declaration / empty

declaration          ::= procedure_declaration
                       / function_declaration
                       / listed_declaration
                       / import_declaration
```

```
                        / module_ declaration
                        / exception_ declaration

statement               ::= assignment_ statement
                        / invocation_ statement
                        / iterative_ statement
                        / flow_ statement
                        / special_ statement
                        / miscellaneous_ statement

import_ declaration     ::= import ID,... from module_ ID
                        / use module_ ID

segment_ list           ::= segment (segment_ ID,...) [aligned (c_ const,...)]
```

Declarations

```
procedure_ declaration  ::= forward [mode] procedure procedure_ ID procedure_ spec

                        / [mode] procedure procedure_ ID procedure_ spec
                                sentence:...
                            endprocedure [|procedure_ ID|]

mode                    ::= inline / fortran / interrupt

procedure_ spec         ::= parameter_ spec [segment_ list]

parameter_ spec         ::= (parameter,...) / ()
                        / param
                              parameter:...
                          endparam

function_ declaration   ::= forward [mode] function function_ ID function_ spec

                        / [mode] function function_ ID function_ spec
                              sentence:...
                            endfunction [|function_ ID|]

function_ spec          ::= parameter_ spec returns variable_ spec [segment_ list]

variable_ spec          ::= variable_ ID,... : [storage] type [initial]

storage                 ::= static
                        / location (l_ constant)
                        / register (register_ spec)
                        / segment_ spec

initial                 ::= initially expression

parameter               ::= ID,... : [call_ type] [storage] type [default] [desc_ clause]
```

segment_ spec ::= **segment** (segment_ ID) [**aligned** (c_ constant)]

desc_clause ::= **with** descriptor_ ID

call_ type ::= **variadic** call_ type
/ [**optional**] [**volatile**] [**in** / **out** / **inout**]
[**ref** / **val**]

default ::= **initially** expression

Type ::= [**different**] [attribute_ list] base_ type [constraint]
[initial] [abstract_ list]

attribute ::= **hidden** / **readonly**
/ **volatile**
/ **packed** / **packed packed** / **unpacked**
/ **c** . **constant bit**

constraint ::= **range** discrete_ type

abstract_ list ::= abstraction / abstraction abstraction_ list

abstraction ::= **starting** [mode] **procedure** procedure_ spec
/ **finishing** [mode] **procedure** procedure_ spec
/ **in** zone_ ID

base_ type ::= basic_ type
/ discrete_ type
/ aggregate_ type
/ special_ type

listed _declaration ::= **declare** (decl)
/ **declare**
decl;...
endeclare

decl ::= variable_ spec
/ constant_ ID,... = l_ constant
/ type_ ID,... **is** [**different**] type [initial]
/ zone_ declaration

basic_ type ::= **integer** / **real** / **logical** / **char** / **long_ real** /
/ **interlock** / **cardinal** / **boolean**

discrete_ type ::= limit..limit
/ '[' enumeration_ ID....']'
/ base_ type

limit ::= expression / ?ID

zone_ declaration ::= zone_ ID : **storage zone** (parameter....)

```
special_type          ::= pointer type
                        / descriptor
                        / general
                        / [mode] procedure procedure_spec
                        / [mode] function function_spec

aggregate_type        ::= array '[' discrete_type,...']' of type
                        / structure
                              field;...
                          endstructure
                        / set of type

field                 ::= fill (c_constant bit)
                        / field_id,... : type
                        / select tag_ID from
                              [case case_label,... : field;...] ...
                          endselect

case_label            ::= c_constant .. c_constant / c_constant

exception_declaration ::= exception exception_ID,...
                        / arm comp_var_ID,...
                        / disarm comp_var_ID,...
```

Statements

```
assignment_statement  ::= expression := expression
                        / expression *= infix_op expression

invocation_statement  ::= procedure_ID ( expression,... ) / procedure_ID ( )
                        / procedure_expression
                        / procedure_ID (parameter_ID: expression,...)
                        / procedure_ID
                              with
                                  parameter_ID: expression;...
                              endwith

procedure_expression  ::= expr_10

iterative_statement   ::= [loop_label:] while boolean_expression do
                                        sentence;...
                                  endwhile [[end_label]]
                        / [loop_label:] repeat
                                        sentence;...
                                  until boolean_expression [[end_label]]
                        / [loop_label:] for for_element do
                                        sentence;...
                                  endfor [[end_label]]
```

```
for_element          ::= for_ID := expression downto expression
                       / for_ID := expression to expression
                       / for_ID := expr then expr while boolean_expr
                       / for_ID in discrete_type
                       / for_ID in set_type

flow_statement       ::=  break label
                       /   loop [loop_label]
                       /   return
                       / [begin_label:] if boolean_expr do
                                            sentence;...
                                        orif boolean_expression do
                                            sentence;...
                                        otherwise sentence;...
                                        endif [[end_label]]

flow_statement       ::= [begin_label:] select expression from
                                            case case_label,... :
                                              sentence;... ...
                                            default : sentence;...
                                        endselect[[end_label]]
                       / [begin_label:] upon viaduct_ID,... leave
                                          sentence;...
                                        through
                                          case viaduct_ID : sentence;... ...
                                        endupon [[end_label]]
                       / via viaduct_ID

special_statement    ::= [begin_label:] region interlock_expression do
                                          sentence;...
                                        otherwise
                                          sentence;...
                                        endregion [[end_label]]
                       / retry
                       / [begin_label:] guard
                                          sentence;...
                                        catch
                                          case exception_ID,... : sentence;...
                                          default : sentence;...
                                        endguard [[end_label]]
                       / raise exception_id [finishing ID,...]
                       / reraise [finishing ID,...]
                       / [begin_label:] block
                                          sentence;...
                                        endblock [[end_label]]

special_statement    ::= [begin_label:] code "machine_designator" do
                                          instruction;...
                                        endcode [[end_label]]
```

```
instruction             ::= assembler_ instruction

misc._ statement        ::= free (pointer_ type_ expression:...)
                          / swap (expression, expression)
                          / assert boolean_ expression
```

Expressions

The numeric values on the "expr" identifiers below represent the operator precedence levels.

```
expression              ::= expr_ 0
                          / when boolean_ expr then expr else expr

expr_0                  ::= [expr_ 0 eqv] expr_ 1
                          / expr_ 0 xor expr_ 1

expr_1                  ::= expr_ 2 [or expr_ 2]

expr_2                  ::= expr_ 3 [and expr_ 3]

expr_3                  ::= [not] expr_ 4

expr_4                  ::= expr_ 5 [relational_ operator expr_ 5]

expr_5                  ::= expr_ 6 [shift_ operator expr_ 6]

expr_6                  ::= [expr_ 6 addition_ operation] expr_ 7

expr_7                  ::= expr_ 8 [multiplication_ operator expr_ 8]

expr_8                  ::= [unary_ sign] expr_ 9

expr_9                  ::= expr_ 10
                          / allocate expr_ 10
                          / force expr_ 10

expr_10                 ::= ID / constant / expr_ 10 (expression....)
                          / expr_ 10 (field_ value....)
                          / (expression)
                          / expr_ 10 '[' expression.... ']'
                          / expr_ 10 . field_ ID
                          / expr_ 10 @
                          / expr_ 10 with parameter_ ID : expression:... endwith

field_ value            ::= field_ ID : expression /
                            '[' case_ element ']' : expression
```

Operators

Infix_ operator	::= **eqv / xor / or / and** / relational_ operator / shift_ operator / addition_ operator / multiplication_ operator
relational operator	::= = /<>/<=/</>=/>
shift_ operator	::= **lsh / rsh**
multiplication_ operator	::= * / '/' / **mod**

Predefined Functions

max	- maximum
min	- minimum
succ	- successor
pred	- predeccessor
abs	- absolute value
round	- real to integer rounded
floor	- largest integer not greater than real
ceiling	- smallest integer not less than real
low	- lower limit of discrete type
high	- upper limit of discrete type
size_ of	- size in bits of data object
descriptor_ of	- descriptor of a type

USCPO-1980: 789-002/5531

Praxis Text Input and Output

Documentation by Bill Nowicki
Septmeber 24, 1980

This document describes a set of simple input and output routines for Praxis under VMS that will be available until a full RMS based I/O package can be completed. This package consists of two modules: Textio, written in Praxis, and Macio, written in Vax assembly language. The following identifiers can be imported from the "Textio" module:

file	Type representing a file
TTY	File for TTY output
TTY_in	File for TTY input
mode	Enumerated type for read, write
read	File open mode
write	" " "
EOF	End of File indicator
EOL	End of Line indicator
Open_file	Open a text file
Close_file	Close a text file
Get_character	Read a single character
Get_integer	Decode an integer
Get_real	Decode a real
Get_padded_string	Read a string
Out_record	Print a line
Out_character	Write a character
Out_string	Write several character
Out_padded_string	Write string without trailing blanks
Out_integer	Write an integer
Out_areal	Write a real
Out_line	Write a string and output the record
TTY_line	Write a message to the terminal

The following procedures can be imported from the MACIO module:

Open	Open a LUN
Close	Close a LUN
Get_record	Actually get the record
Put_record	Actually put the record

The routines use a line buffer of 132 characters. The special character constant EOF is defined as a flag character returned when end of file is reached. When a file is opened, the second parameter is a mode which is either read, or write. A file is a structure which contains a logical unit number (LUN), a buffer position counter, and a line buffer. The files "TTY" and "TTY_in" are special static files that can be used for output and input to the terminal. They are initially set to the logical device "Sys$Input" for

input and "Sys$Output" for output, which VMS associates the user's terminal. I/O to the terminal is immediate, instead of buffered, so that a prompt string sent to TTY followed by a read from TTY_in will have the expected effects. The package allocates LUNS starting at one, finding a free lun for each open call. Normally the user should not be concerned about this. Close_file releases the luns so that they can be "recycled". A maximum of 8 files can be open at any time.

Currently to access these routines you must define the logical symbol PRXSLIBRARY before you compile your program. This is normally done in [CCLIB]SYMBOLS. Just include an appropriate import statement to import the routines that you want to use. Usually you will import only from Textio.

The object modules should be automatically found by the linker, since the symbol LNK$Library is defined by the command "UsePrxLib".

procedure Open_file(f: inout ref file, m:mode, name:in ref array[1..?N] of char)

This procedure allocates a logical unit number, opens the named file for either "read" or "write", and resets the buffers for the file. Currently eight files can be opened in addition to TTY and TTY_in. The file name can be a usualy VMS name specification, usually as a quoted string.

procedure Close_file(f: inout ref file)

This procedure closes a file and releases the associated lun.

function Get_character(f: inout ref file) returns c: char

This function returns the next character from the specified file. If the end of the file is reached, the character EOF is returned. A new record is read when needed, and the EOL character is returned.

function Get_integer(f: inout ref file) returns N: integer

This function returns a decimal integer read from the specified text file. Spaces, tabs, and formfeeds are skipped before the number.

function Get_real(f: inout ref file) returns X: real

This function returns a real number read from the specified text file. A real number consists of zero or more digits followed by a decimal point, followed by zero or more digits. Thus no exponential notation like Fortran's is currently implemented. Spaces, tabs, and formfeeds are skipped before the number.

```
procedure Get_padded_string(f: inout ref file, s: inout ref array [1..?n] of
    char)
```

This procedure returns a string read from the specified text file. The string is read up to and end of line, and the rest of the string is padded with blanks. The first character of the string is set to EOF when the end of file is reached.

```
procedure Out_record(f:  inout ref file)
```

This procedure outputs the current record (line) to the text file, and resets the buffer position.

```
procedure Out_character(f:  inout ref file, c:  char)
```

This procedure outputs a character to a file. In reality it just puts the character into a buffer, and Out_record must be called to actually write the record to the file. On immediate I/O files, like TTY, the I/O is done immediately.

```
procedure Out_string(f:  inout ref file, s:in ref array[1..?N] of char)
```

This procedure writes the string to the specified file. It is usually used with a quoted string constant as second parameter.

```
procedure Out_padded_string(f: inout ref file,
                    s:in ref array[1..?N] of char)
```

This procedure is similar to Out_string, but it does not print trailing blanks or nulls in the string. For example, Out_string(TTY, "abc ") will print six characters, while Out_padded_string(TTY, "abc ") will only print three.

```
procedure Out_integer(f:  inout ref file, N:  integer)
```

This procedure puts an integer to a text file as a string of decimal digits. If the integer is negative, a minus sign is printed before it. No leading or trailing blanks are printed.

```
procedure Out_areal(f:  inout ref file, X:  real)
```

This procedure puts a real number to a text file as a string of decimal digits. If the real number is negative, a minus sign is printed before it. No leading or trailing blanks are printed.

```
procedure Out_line(f:  inout ref file, s:in ref array[1..?N] of char)
```

This procedure puts a string to the given file, and writes the record to the file. It is equivalent to a call to Out_string followed by a call to Out_record.

```
procedure TTY_line( s:in ref array[1..?N] of char)
```

This procedure puts a string to the TTY file, and writes the record to TTY. It is equivalent to a call to Out_string followed by a call to Out_record, with TTY as a file parameter. This routine is used to print quick one-line error messages, for example.

---------- The following are interal routines in MACIO ----------

```
procedure Open(lun:integer, mode:integer, name:in ref array[1..?N] of char)
```

This is the Macro routine which uses RMS to actually open a file on a given LUN.

```
procedure Close(lun:  integer)
```

This is the Macro routine to close the given Logical Unit Number.

```
procedure Get_record(lun: integer, buffer: inout ref array[0..?N] of char,
                     count: inout ref integer)
```

This is the Macro routine that actually executes the RMS calls to fill the record buffer, from the file associated with the given Logical Unit Number. It returns the number of characters read in the "count" parameter.

```
procedure Put_record(lun: integer, buffer:in ref array[0..?N] of char,
                     count: integer)
```

This is the Macro routine that actually executes the RMS calls to write the record buffer, of the given length, to the file associated with the given Logical Unit Number.

```
enddocument
```

1.4 Global Declaration Files For Arrays

Declaration of arrays in FSM source files is described in Section 4.1. In addition, the user must provide a separate Praxis declaration file for any emulation that uses arrays. The Praxis source code of this file, under the specific name AMRFGBL.PRX, should be compiled and placed in the HCSE_LIBRARY. The parse will require the file AMRFGBL.OBJ and AMRFGBL.SPS when arrays are declared. The Praxis source code contains only a single declaration section which exports array parameters for each array employed. For a single array declared in the FSM source file by statement of the form

```
//(input,inparameter,output,outparameter) (space)
   ARRAYOF_indexname_basetype
```

the file AMRFGBL.PRX would consist of the following text:

```
module amrfgbl

    export indexname_max,indexname_range
    export arrayof_indexname_integer
    export arrayof_indexname_real
    export arrayof_indexname_char
    export arrayof_indexname_string
    export arrayof_indexname_boolean
    declare
        pchar is char initially $<NUL>
        string is packed array [1..32] of pchar
        indexname_max = (constant equal to maximum index
           value)
        indexname_range is integer range 1..indexname_max
        indexname_dimension is integer range 0..indexname_max
        arrayof_indexname_integer is array [indexname_dimen]
           of integer
        arrayof_indexname_real is array [indexname_dimen]
           of real
        arrayof_indexname_char is array [indexname_dimen]
           of char
        arrayof_indexname_string is array [indexname_dimen]
           of string
        arrayof_indexname_boolean is array [indexname_dimen]
           of boolean
    enddeclare

endmodule {amrfgbl}
```

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBSIR 85-3156	2. Performing Organ. Report No.	3. Publication Date May 1985

4. TITLE AND SUBTITLE

Hierarchical Control System Emulation
User's Manual

5. AUTHOR(S)

Cita Furlani (Editor)

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

10. SUPPLEMENTARY NOTES

Previously published as NBS-GCR-82-413 - NTIS PB83-141952

[X] Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

The Hierarchical Control System Emulation is a collection of computer programs written in the high-level Praxis language for use on a Digital Equipment Company VAX 11/780™ processor under the VMS™ operating system. These programs allow the user to write, debug, and concurrently emulate modules of a hierarchical control system and to simulate the physical plant which is controlled. The emulation executes in real time and interactive display and data logging capabilities are included. The emulation is intended as a computer-aided control system design tool for the NBS Automated Manufacturing Research Facility. The User's Manual describes the use of the emulation and provides necessary theoretical background; it is not application-specific.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

Automated manufacturing; automatic control; hierarchical control systems; computer-aided design; computer-aided manufacturing; simulation

13. AVAILABILITY

[X] Unlimited

[ ] For Official Distribution. Do Not Release to NTIS

[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

136

15. Price

$14.50

USCOMM-DC 6043-P80

FEDERAL INFORMATION PROCESSING STANDARD SOFTWARE SUMMARY

01. Summary date
Yr. 85 | Mo. | Day

02. Summary prepared by (Name and Phone)
Cita Furlani, 921-2461 area code (301)

03. Summary action
New [X] Replacement [ ] Deletion [ ]
Previous Internal Software ID

04. Software date
Yr. 85 | Mo. 11 | Day 15

05. Software title
Hierarchical Control System Emulation User's Manual

06. Short title HCSE

07. Internal Software ID

08. Software type
- [X] Automated Data System
- [ ] Computer Program
- [ ] Subroutine/Module

09. Processing mode
- [X] Interactive
- [ ] Batch
- [ ] Combination

10. Application area

General
- [ ] Computer Systems Support/Utility
- [X] Scientific/Engineering
- [ ] Bibliographic/Textual
- [ ] Management/Business
- [ ] Process Control
- [ ] Other

Specific
Industrial Control Design

11. Submitting organization and address
U.S. Department of Commerce
National Bureau of Standards
Bldg. 220 - Room A-127
Gaithersburg, MD 20899

12. Technical contact(s) and phone
Cita Furlani 921-2461 area code (301)

13. Narrative
The Hierarchical Control System Emulation is a collection of computer programs written in the high-level Praxis language which allow the user to write, debug, and concurrently emulate modules of a hierarchical control system and to simulate the physical plant which is being controlled. The emulation executes in real time, and interactive display and data logging facilities are included. It is intended as a computer-aided design tool for the NBS Automated Manufacturing Research Facility shop floor control system.

14. Keywords
Computer-aided manufacturing; hierarchical control system design; simulation; emulation; automation; industrial control

15. Computer manuf'r and model	16. Computer operating system	17. Programing language(s)	18. Number of source program statements
DEC VAX 11-780	VMS, Vers. 2.7	Praxis, Fortran	

19. Computer memory requirements	20. Tape drives	21. Disk/Drum units	22. Terminals
1 Mbyte	None	System disk required.	VT52, VT100 or equivalent

23. Other operational requirements
None

24. Software availability
Available [ ] Limited [X] In-house only [ ]
For government use only.

25. Documentation availability
Available [X] Inadequate [ ] In-house only [ ]
NTIS

26. FOR SUBMITTING ORGANIZATION USE